Pasi Tiihonen

# Pelinkehitys Unity3D-pelimoottorilla aloittelijoille

Opinnäytetyö

Kevät 2013

Tekniikan yksikkö

Tietotekniikan koulutusohjelma

Seinäjoen ammattikorkeakoulu
SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

SEINÄJOEN AMMATTIKORKEAKOULU

## Opinnäytetyön tiivistelmä

Koulutusyksikkö: Tekniikan yksikkö

Koulutusohjelma: Tietotekniikan koulutusohjelma

Suuntautumisvaihtoehto: Ohjelmistotekniikka

Tekijä: Tiihonen, Pasi

Työn nimi: Pelinkehitys Unity3D –pelimoottorilla aloittelijoille

Ohjaaja: Lahti, Markku

Vuosi: 2013 Sivumäärä: 56 Liitteiden lukumäärä: 4

Opinnäytetyössä kehitettiin yksinkertainen tietokonepeli Unity3D-pelimoottorilla ja apuna käytettiin ilmaista Blender-ohjelmaa 3D-mallinnuksessa sekä Photoshop-kuvankäsittelyohjelmaa mallien suunnittelussa ja tekstuurien piirtämisessä. Työn tarkoitus on ohjeistaa ensikertalaista käyttämään Unity-moottoria apuna pelin kehittämisessä.

Pelin toiminnallisuus ohjelmoitiin C#-kielellä ja työ olettaa, että lukijalla on valmiiksi kokemusta ohjelmoinnista. Työssä käsitellään myös 3D-mallinnusta, jolla pelin hahmot ja objektit tehdään. Opinnäytetyön tavoitteena on antaa perusteellinen ohjeistus pelin kehittämisestä aloittelijoille, jotka haluavat käyttää Unity-moottoria kehityksen apuna.

Avainsanat: Unity, Blender, Photoshop, pelisuunnittelu, peli, C#

SEINÄJOKI UNIVERSITY OF APPLIED SCIENCES

## Thesis abstract

Faculty: School of Technology

Degree programme: Information Technology

Specialisation: Software Engineering

Author/s: Tiihonen, Pasi

Title of thesis: Game Development with Unity3D for beginners

Supervisor(s): Lahti, Markku

Year: 2013 Number of pages: 56 Number of appendices: 4

In this thesis a simple computer game was developed with Unity3D game engine. 3D-modelling was done with Blender and Photoshop was used to design models and draw the textures. The meaning of this work was to instruct beginners to use Unity when developing games.

The game's logic was programmed with C#-language and it was presumed that the reader is already familiar with programming. The thesis also studied 3D modelling when making the characters and objects in the game. The thesis aimed to give a comprehensive tutorial on game development for beginners who want to use Unity engine as an aid.

Keywords: Unity, Blender, Photoshop, video game, C#

# SISÄLTÖ

# Kuvio- ja taulukkoluettelo

## Käytetyt termit ja lyhenteet

| | |
|---|---|
| **3D-malli** | Kolmiulotteinen esitys esineestä tai hahmosta näytöllä. |
| **Debuggaus** | Virheiden etsiminen koodista. |
| **Indiekehittäjä** | Itsenäinen pelinkehittäjä, joka ei ota vastaan rahoitusta julkaisuyhtiöiltä. (Engl. independent developer.) |
| **Pelihahmo** | Pelin sisäinen ”elollinen” objekti, joka liikkuu ja on vuorovaikutuksessa peliympäristön kanssa. |
| **Pikavalikko** | Ponnahdusvalikko, joka tulee esille hiiren oikealla painikkeella. |
| **Skripti** | Komentosarja, jolla kerrotaan peliobjektien toiminnallisuus. Komentosarjat ovat pelin koodi. |
| **Valikkokomento** | Polku, joka kuvaa työkaluvalikon vaihtoehtojen valitsemista. |

# 1 JOHDANTO

## 1.1 Työn tausta

Tietokonepelien kehittäminen ja rakentaminen tyhjästä vaatii hyvää ohjelmointitaitoa ja vie aikaa. Nykyiset pelit ovat kuitenkin hyvin suuria ja uusien pelien tulisi valmistua vain muutamassa vuodessa. Tästä syystä suurin osa pelitaloista rakentaa pelinsä *pelimoottoreilla*. Pelitalot tekevät usein pelimoottorinsa itse, mutta monet käyttävät markkinoilta jo löytyviä moottoreita.

Pelimoottori on ohjelma, joka nopeuttaa pelin kehitystä huomattavasti verrattuna siihen, että pelin kaikki ominaisuudet kirjoitettaisiin alusta lähtien. Pelimoottoreihin on tyypillisesti kirjoitettu valmiiksi fysiikanmallinnus, grafiikan piirtäminen ja komentojen syöttäminen (Ward 2008, 1).

Vaikka pelimoottori sisältääkin paljon pelien perusmateriaaleja, itse sisältö, ulkonäkö ja se, miten pelin objektit reagoivat toisiin, muodostavat pelin. Nämä asiat on kehittäjän itse tuotettava. (Ward 2008, 1.)

Unity3D on Unity Technologiesin valmistama pelimoottori, jonka suosio on kasvanut suurimpien pelimoottorien rinnalle vain muutamassa vuodessa. Vaikka Unity ei visuaalisesti ole vielä täysin samalla tasolla kuin kilpailijansa CryEngine tai UDK, se on kilpailukykyinen hintansa ja laajan alustatukensa ansiosta. (Unity Technologies 2013a.)

Unity-moottorin osien logiikka voidaan ohjelmoida kolmella eri kielellä, jotka ovat *JavaScript*, *C#* ja *Boo.* Tässä työssä kaikki logiikka kirjoitetaan C#-kielellä.

## 1.2 Työn tavoite

Työn tavoite on kehittää yksinkertainen kolmiulotteinen tietokonepeli Unity-moottorilla ja esittää pelin tekovaiheet yksityiskohtaisesti. Työn tarkoitus on toimia

ohjeena aloitteleville pelintekijöille, joilla ei ole kokemusta Unity-moottorista tai 3D-mallintamisesta.

### 1.3 Työn rakenne

Toinen luku kertoo yleisesti pelien suunnittelun ja toteutuksen vaiheista ja syistä. Lisäksi luvussa vertaillaan kolmea suosituinta pelimoottoria ja perustellaan Unity3D-moottorin valintaa.

Kolmannessa luvussa tutustutaan Unity-moottorin käyttöliittymään ja kehitysympäristössä liikkumiseen. Lisäksi tehdään harjoitus peliobjektien lisäämisestä ja muokkaamisesta, valaistuksesta, materiaaleista ja fysiikoista. Lopuksi luodaan Unityn oma valmis ohjattava pelihahmo.

Neljäs luku käsittelee itse pelin tekemistä, ja se alkaa uuden projektin luomisella. Apuna käytetään toisen luvun tietoja ja uutena asiana esitellään Script-elementtien luominen ja niiden liittäminen peliobjekteihin. Luvussa luodaan pelaajan ohjaama päähahmo.

Viidennessä luvussa tutustutaan Blender-ohjelmaan ja sen avulla mallinnetaan kenttä, jossa pelaaja liikkuu.

Kuudennessa luvussa suunnitellaan ja luodaan vihollinen pelaajalle. Luonnissa sovelletaan kolmannen ja neljännen luvun tietoja sekä tutustutaan skripteihin tarkemmin.

Seitsemäs luku esittelee pelin pysäyttämisen ja käyttöliittymän piirtämisen kaksiulotteisilla grafiikkaelementeillä.

Kahdeksas luku on yhteenveto työn tavoitteista ja vaiheista. Luvussa arvioidaan, kuinka hyvin tavoitteisiin päästiin.

# 2 VIDEOPELIEN SUUNNITTELU JA TOTEUTUS

Videopelien kehitys vaatii monenlaista osaamista. Mielikuvituksen ja kertomisen taidon lisäksi pelien tekijöiltä vaaditaan teknistä osaamista ja taiteellista näkemystä. (Crosby 2011, 1.)

## 2.1 Suunnittelu

Videopeli suunnitellaan samankaltaisesti kuin mikä tahansa visuaalinen taideteos. Suunnittelu on syytä tehdä ennen pelin rakentamista, sillä se toimii koko projektin pohjana. Pelin yleinen graafinen tyyli ja laji ovat tärkeimpiä ominaisuuksia, jotka tulisi päättää ennen projektin aloittamista.

Pelihahmojen suunnittelu on tärkeä osa pelin kehitystä. Hahmojen on oltava mielenkiintoisia, sillä pelaajat ohjaavat niitä tai ovat niiden kanssa vuorovaikutuksessa. Joissain tapauksissa hahmojen suunnittelijat piirtävät hahmoista kuvia, joiden pohjalta hahmojen 3D-mallit tehdään. (Crosby 2011, 2.)

## 2.2 Toteutus

Pelin voi rakentaa joko aputyökaluja käyttämällä tai alusta lähtien ohjelmoimalla. Aloittelijoille ja ammattilaisille on olemassa lukuisia pelimoottoreita, joiden avulla pelejä voi tuottaa ilman valtavaa ohjelmointiurakkaa. Valmiit pelimoottorit helpottavat pelituotantoa, koska niihin on ohjelmoitu valmiiksi raskaat matemaattiset algoritmit ja niissä on yleensä graafinen käyttöliittymä helpottamassa kehitystä. (Crosby 2011, 3.)

Täysin ainutlaatuisen pelin tekeminen vaatii lähes poikkeuksetta oman koodin kirjoittamista. Pelien tekemisessä ohjelmointikoodin kirjoittaminen ja lukeminen ovat erittäin hyödyllisiä. Monimutkaisten pelien kehittäminen vaatii tyypillisesti monimutkaista koodia. (Crosby 2011, 4.)

## 2.3 Unity3D-, CryEngine- ja UDK-pelimoottorit

Aloittelevalle pelinkehittäjälle pelimoottorit ovat erinomainen vaihtoehto, sillä niiden avulla kehittäjä voi keskittyä alusta lähtien pelin sisällön tuottamiseen. Kehittäjäyhtiöt, kuten CryTek ja Epic Games ovat lanseeranneet omat pelinkehitysympäristönsä markkinoille. Näiden pelimoottoreiden käyttö on ilmaista siihen asti, kunnes niillä kehittämäänsä peliä alkaa myydä. Tällöin tietty prosentti pelin tuotosta on maksettava pelimoottorin kehittäjäyhtiölle.

**UDK.** UDK on Epic Games-yhtiön pelimoottori, joka on ilmainen opetus- ja ei-kaupalliseen käyttöön (Epic Games 2013). UDK-moottorin kaupallinen lisenssi maksaa 99 dollaria ja sen jälkeen yli 5000 dollarin tuloista 25 % maksetaan Epic Games-yhtiölle.

**CryEngine.** CryEngine on CryTek-yhtiön valmistama pelimoottori, joka on tästä kolmikosta graafisesti näyttävin. UDK-moottorin tapaan CryEngine-pelimoottorista on tarjolla ilmainen versio ei-kaupalliseen käyttöön ja kaupallinen lisenssi. (CryTek 2013.)

**Unity3D.** Unity on pelimoottorikolmikosta graafisesti alkeellisin. Moottori on kuitenkin aloittelijalle helpoiten lähestyttävä vaihtoehto käyttöliittymänsä ansiosta. Lisäksi sen käyttö on halvempaa kiinteän maksutyylin ansiosta.

Unity3D-pelimoottoria myydään kahtena versiona. Riisuttua versiota voi käyttää ilmaiseksi ja sillä tehtyjä pelejä myydä vapaasti. Unity Pro -versio maksaa 1500 dollaria ja maksu sisältää kaikki tulevat päivitykset. Jokaisen kehittäjän on ostettava erikseen Unity Pro -versio. (Unity Technologies 2013b.)

Kun pelimoottoreilla tehtyjä pelejä myydään, moottoreiden valmistajat pyytävät osaa pelien myyntituloista. Pelinkehittäjien tulee siis valita pelimoottorinsa tarkasti voittojensa maksimoimiseksi.

Jos viiden kehittäjän yhtiö tekee UDK-moottorilla pelin, jonka myyntitulot ovat 20 000 dollaria, yhtiölle jää voitolle 16 151 dollaria. Jos yhtiö käyttäisi Unity3D-

moottoria, jonka hinta on jokaiselle viidelle kehittäjälle erikseen 1500 dollaria, yhtiö jäisi voitolle vain 12 500 dollaria.

Jos pelin myyntitulot olisivat 50 000 dollaria, Unity-pelimoottori olisi selvästi edullisempi vaihtoehto. Tämän lisäksi on huomioitava, että Unity-moottorin alustatuki on UDK-moottoria laajempi. Unity-kehittäjä voisi siis myydä peliään laajemmalle käyttäjäkunnalle.

## 2.4 Unity3D indiekehittäjän valintana

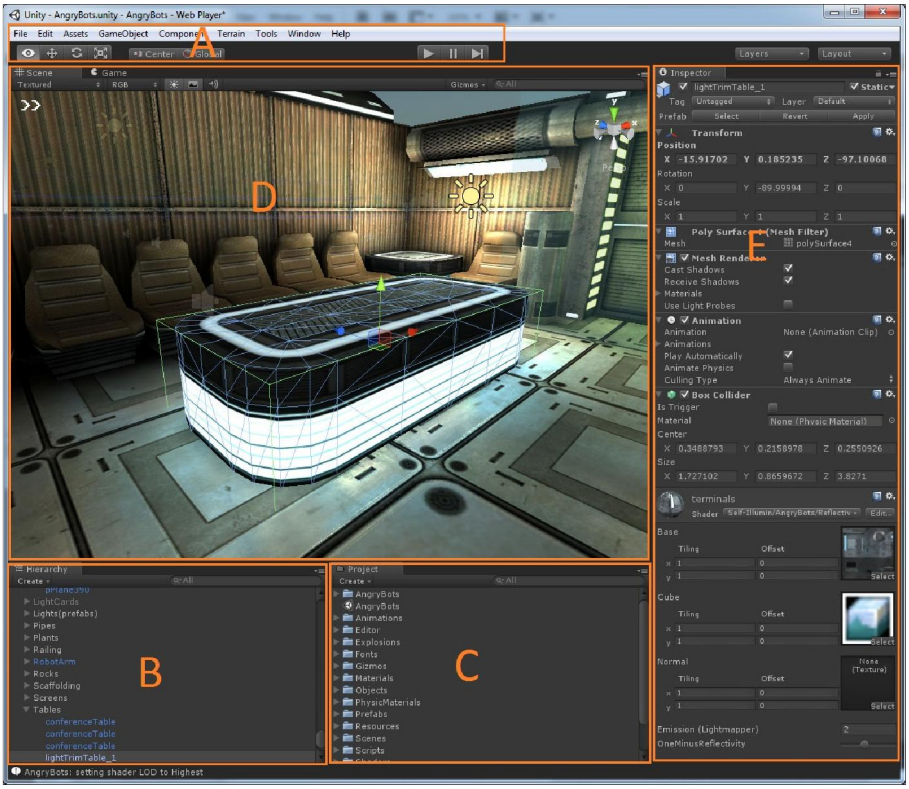
Unity-pelimoottoria ei ole tarkoitettu suurien pelien kehittämiseen, eikä sillä ole kilpailijoidensa veroisia visuaalisia ominaisuuksia. Unity on kuitenkin hintansa ja laajan alustatukensa ansiosta itsenäisten pelinkehittäjien suosiossa. (McKleinfeld 2012.)

Unity on suunniteltu toimimaan Blender-mallinnusohjelman kanssa. Blender-ohjelmalla rakennetut mallit voi tallentaa suoraan Unity-projektin kansioon, josta Unity lukee ne automaattisesti. Unity-moottorin mekaniikkojen kirjoittamiseen voi käyttää C#-kieltä. (Unity Technologies 2013c.)

Tässä työssä Unity on valittu kehitysympäristöksi helpon käyttöliittymänsä ja C#-tukensa vuoksi.

# 3 UNITYN KÄYTTÖLIITTYMÄ

Unityn käyttöliittymä (kuvio 1) koostuu työkaluista (kohta A) sekä *Hierarchy*- (kohta B), *Project*- (Kohta C), *Game*-, *Scene*- (kohta D), *Console*- ja *Inspector*-näkymistä (kohta E). Näillä hallitaan *peliobjekteja* (GameObject), jotka toimivat rakennuspalikoina kaikille pelin ominaisuuksille. Peliobjekteja ovat mm. esineet, hahmot, valot ja äänet. Peli rakennetaan *kentistä* (Scene), joita voi pitää esillä yksi kerrallaan.



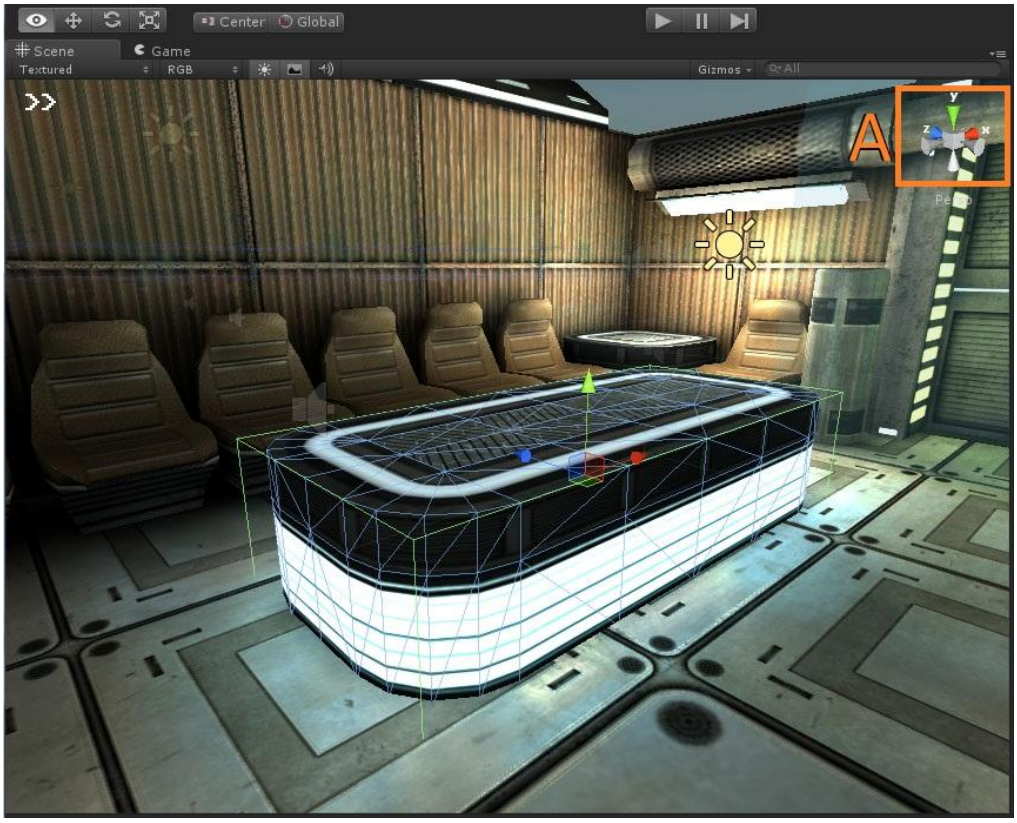

Kuvio 1. Käyttöliittymä

## 3.1 Navigointi

Kentän kaikki objektit näkyvät Scene- ja Game-näkymissä Scene-näkymässä on vapaasti ohjattava kamera, jolla voi liikkua kentässä hiiren oikean painikkeen ollessa pohjassa. Kameraa ohjataan eteen, taakse ja sivuille W-, S-, A-, D-, Q- ja E-näppäimillä ja hiirellä ohjataan suuntaa, johon kamera osoittaa. Game-

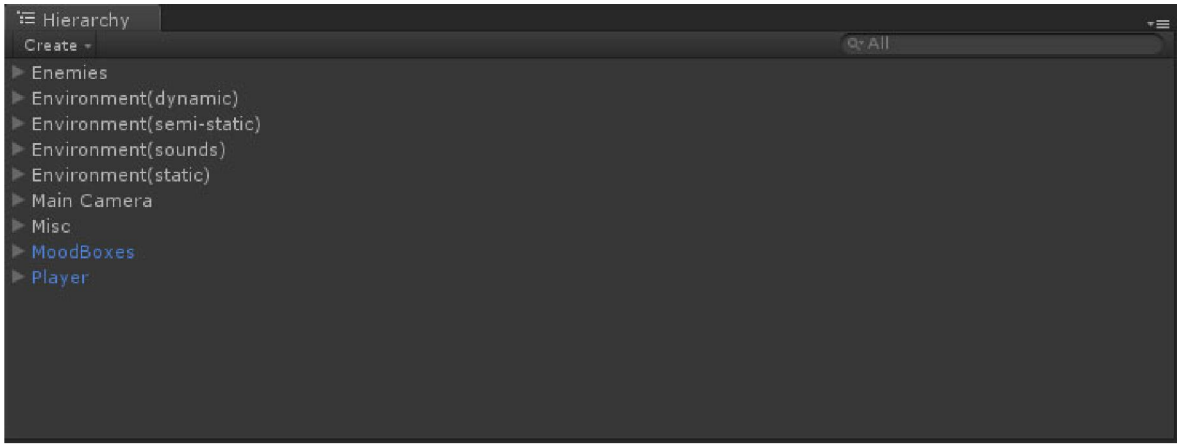
näkymässä ei voi liikkua, sillä sen kameranäkymä on lukittu kentän pääkameraan. Näkymä näyttää kentän tilan käynnistyshetkellä.

Scene-näkymän (kuvio 2) oikeassa ylänurkassa on kolmiulotteinen ohjain (kohta A), joka näyttää koordinaattien suunnat ikkunassa. Painamalla jotain ohjaimessa olevaa kartiota, kuvakulma siirtyy osoittamaan kartion kärjen suuntaan. Painamalla ohjaimen keskellä olevaa kuutiota voi vaihtaa kameran näkymän *perspektiiviseksi* tai *ortogonaaliseksi*. Perspektiivinen näkymä näyttää kaiken normaalisti. Ortogonaalinen näkymä näyttää kentän objektit samankokoisina etäisyyksistä riippumatta. (Unity Technologies 2013d.)



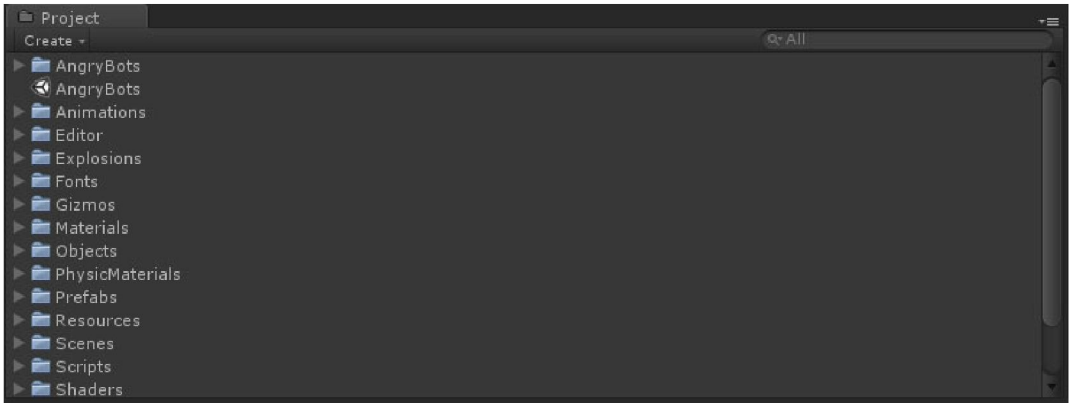

Kuvio 2. Scene-näkymä

**Hierarchy-näkymä.** Hierarchy-näkymässä (kuvio 3) on listattu kaikki kentän peliobjektit. Peliobjektit ovat hierarkisessa järjestyksessä. Yhdellä peliobjektilla voi olla monta aliobjektia eli *lasta,* mutta korkeintaan yksi *vanhempi*. Tämä hierarkia mahdollistaa peliobjektien ryhmittelyn: kun vanhempaa siirretään, sen lapset siirtyvät myös.




Kuvio 3. Hierarchy-näkymä

**Project-näkymä.** Project-näkymä (kuvio 4) listaa kaikki projektiin kuuluvat tiedostot ja kansiot. Tiedostoja ovat esimerkiksi pelihahmojen ja esineiden mallit, äänet, komentosarjat ja *valmisobjektit* (prefab). Tästä näkymästä voi lisätä kenttään peliobjekteja ja peliobjekteihin komponentteja raahaamalla.

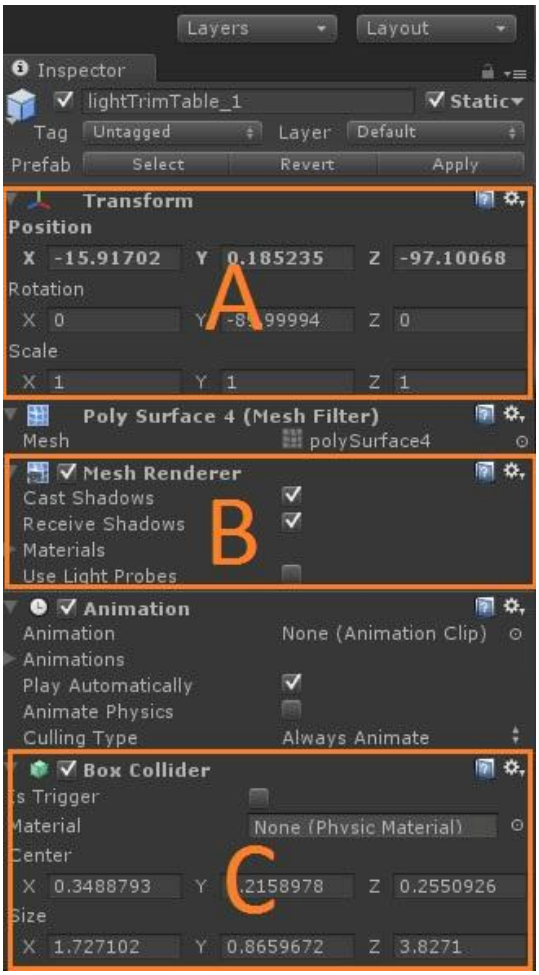



Kuvio 4. Project-näkymä

**Inspector-näkymä.** Inspector-näkymä (kuvio 5) näyttää valitun objektin tiedot ja komponentit sekä komponenttien tiedot. Kaikilla peliobjekteilla on pakollinen Transform-komponentti (kohta A), mutta muut komponentit ovat vaihtoehtoisia. Muita komponentteja ovat esimerkiksi objektin malli (kohta B) ja törmäysrajat (kohta C). Transform-komponentti kertoo peliobjektin sijainnin suhteessa objektin vanhempaan. Jos objektilla ei ole vanhempaa, objektin sijainti kerrotaan suhteessa kentän nollapisteeseen. Peliobjekteihin voi lisätä komponentteja

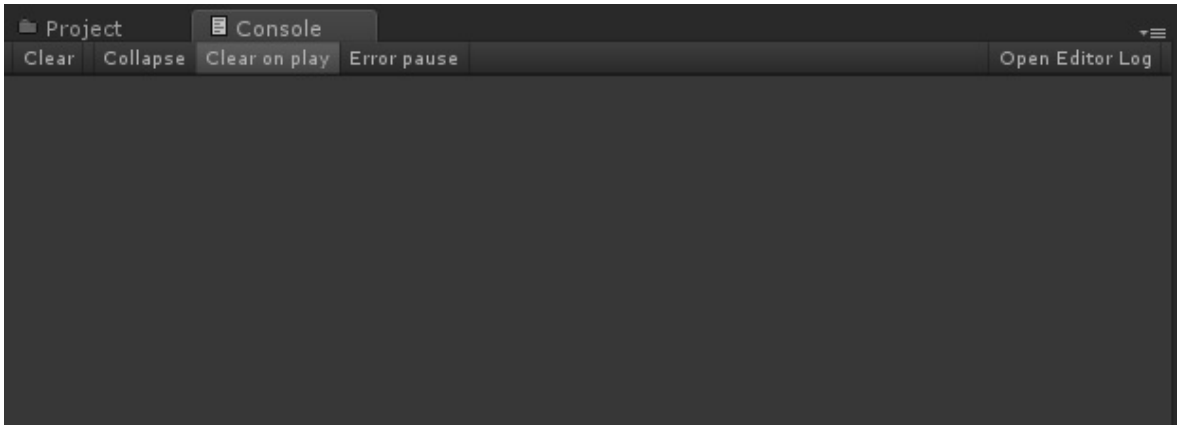
työkaluvalikosta tai vetämällä ne hiirellä Project-näkymästä Inspector-näkymään. Komponentit voi poistaa pikavalikon **Delete**-komennolla.

Inspector-näkymä voi näyttää muidenkin kuin peliobjektien tietoja. Siitä voi vaihtaa esimerkiksi pelin grafiikka- ja fysiikka-asetuksia.




Kuvio 5. Inspector-näkymä

**Console-näkymä.** Tässä näkymässä (kuvio 6) näkyy pelin ajon aikana näkyvät käyttäjän lähettämät viestit, virheet ja varoitukset. Se toimii apuna, kun skriptien toiminnallisuuksista etsitään virheitä.




Kuvio 6. Console-näkymä

### 3.2 Peliobjektin luominen ja käsittely

Kaikki toiminta Unityn peleissä perustuu peliobjekteihin. Peliobjekti luodaan valitsemalla **GameObject → Create Empty**. Unity lisää tyhjän peliobjektin kenttään ja antaa sille nimen ”GameObject”. Objekti on nyt listattuna Hierarchy-näkymässä ja sen tietoja voi tarkastella Inspector-näkymässä. Tässä vaiheessa objektilla on vain Transform-komponentti, joka kertoo objektin sijainnin suhteessa kentän nollapisteeseen.

Tyhjää peliobjektia voi käyttää esimerkiksi muiden objektien ryhmittelyyn. Tyhjä peliobjekti asetetaan valituille peliobjekteille vanhemmaksi, mikä selkeyttää Hierarchy-näkymän lukemista. Objektia liikuttaessa myös sen lapset liikkuvat.

**Valmiin peliobjektin luominen.** Unity-moottorissa on valmiina yksinkertaisia peliobjekteja, joita kannattaa käyttää apuna pelin rakentamisessa. Valmiita objekteja ovat mm. geometriset muodot, kuten kuutiot, pyramidit ja sylinterit sekä kamerat ja valon ja äänen lähteet. Uusi kuutio tehdään valitsemalla **GameObject → Create Other → Cube**. Kameran voi kohdistaa kuutioon **F**-näppäimellä.

**Peliobjektin liikuttelu ja muokkaaminen.** Kuution keskipisteessä on kolmen nuolen muodostama kontrolleri, jolla kuutiota voi siirrellä. Kuutiota voi myös pyöritellä ja sen mittasuhteita voi muuttaa valitsemalla siihen tarkoitetun kontrollerin. Pyörityskontrollerin saa esille **E**-näppäimellä ja mittasuhdekontrollerin **R**-näppäimellä. Takaisin siirtelykontrolleriin pääsee **W**-näppäimellä.

**Materiaalin lisääminen peliobjektiin.** Materiaali on tieto siitä, miltä peliobjektin grafiikka näyttää. Uusilla peliobjekteilla on aina käytössään oletusmateriaali. Materiaali voi olla läpinäkyvä, osittain läpinäkyvä, läpinäkymätön ja/tai itsevalaiseva. Materiaalin tietoihin kuuluvat valon käyttäytyminen ja *tekstuuri*. Tekstuuri on kaksiulotteinen kuva, johon kolmiulotteisen objektin voi ”kääriä”. (Unity Technologies 2013e.)

Materiaalin voi vaihtaa etsimällä objektin **Mesh renderer** -komponentin Inspector-näkymästä objektin ollessa valittuna. **Materials**-otsikon alta löytyvät **size**- ja **Element** *0*-vaihtoehdot, joista jälkimmäisessä on oletuksena **Default-Diffuse**-materiaali. Painamalla sen kohdalla oikeassa reunassa olevaa ympyrää avautuu valintaikkuna, josta peliobjektille voi valita uuden materiaalin.

**Fysiikkakomponentin lisääminen peliobjektiin.** Osa pelissä olevista objekteista voi olla fysiikan lakien alaisena. Jos objektilla ei ole fysiikanmallinnusta, se ei reagoi muihin pelin objekteihin. Peliobjekti saadaan tottelemaan fysiikan lakeja *Rigidbody*-komponentilla. Komponentti lisätään valikkokomennolla **Component → Physics → Rigidbody**. Komponentti näkyy Inspector-näkymässä, jossa sille on asetettu oletusarvot. Objektilla on nyt massa ja siihen vaikuttavat voimat, kuten painovoima.

**Valojen luonti.** Valot ovat peliobjekteja, joilla voi luoda peliin haluamansa tunnelman. Valoja on Unity3D-moottorissa neljä eri tyyppiä: suuntavalo, pistevalo, spottivalo ja aluevalo.

- **Suuntavalo** (Directional Light)**.** Suuntavalo loistaa valoa yhteen suuntaan tasaisesti kaikkialle kenttään. Sitä käytetään pääasiassa simuloimaan auringon tai kuun tuottamaa valoa.
- **Pistevalo** (Point Light)**.** Pallon muotoinen valo, joka valaisee kaikkiin suuntiin. Pistevalot ovat yleisimpiä valoja videopeleissä ja niitä käytetään esimerkiksi räjähdyksissä ja lampuissa.
- **Spottivalo** (Spotlight)**.** Valo, joka loistaa vain yhteen suuntaan kartiomaisesti. Spottivalot toimivat hyvin taskulamppuina ja auton etuvaloina.

- **Aluevalo** (Area light)**.** Valo, joka lähtee neliön muotoisen tason toiselta puolelta. Näitä valoja ei voi käyttää reaaliaikaisesti, sillä ne ovat liian raskaita. (Unity Technologies 2013f.)

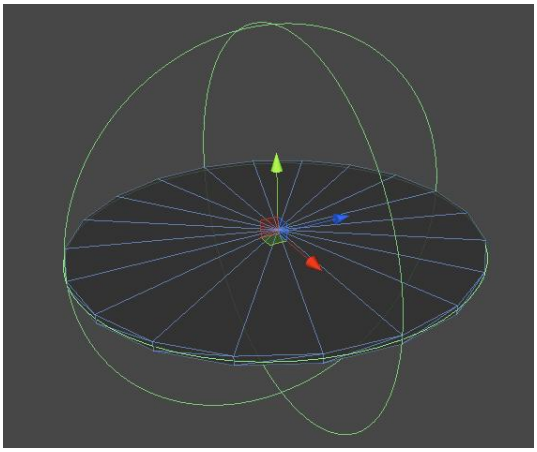
Valot luodaan valitsemalla **GameObject → Create Other →** (jokin valotyypeistä).

# 4 UUDEN PROJEKTIN ALOITTAMINEN

Unity-moottorissa peliä käsitellään projektina, joka koostuu kentistä. Opinnäytetyössä tehdään esimerkkiprojekti. Projektin tekemisessä käytetään apuna edellisten lukujen tietoja. Uusi projekti luodaan valitsemalla **File → New Project**. Ruutuun avautuu ikkuna, jossa uuden projektin voi nimetä ja sille voi tuoda valmiita paketteja. Esimerkkiprojektia varten tarvitaan vain **Skyboxes.unitypackage**. Projekti luodaan **Create**-komennolla, jolloin Unity käynnistyy uudestaan ja lataa uuden projektin tyhjällä kentällä.

## 4.1 Pelitason luonti

Tyhjään kenttään tehdään alusta, jolla pelaaja voi liikkua, valitsemalla **GameObject → Create Other → Cylinder**. Luotu sylinteri sijoitetaan kentän keskelle vaihtamalla sen **Transform**-komponentin **Position**-koordinaatit nolliksi. Lisäksi muutetaan sen mittasuhteita kiekkomaiseksi antamalla **Scale**-arvoiksi **10; 0,1; 10** (Kuvio 7).



Kuvio 7. Sylinteri muutettu alustaksi.

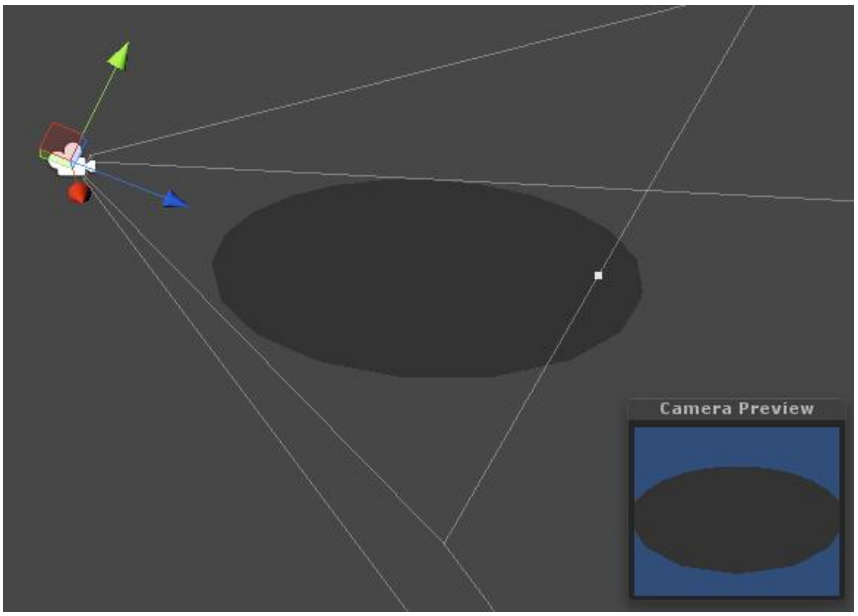
Valitun sylinterin ympäri kulkevat vihreät viivat kertovat, missä sen *törmäysrajat* (collider) sijaitsevat. Nykyiset törmäysrajat ovat väärät, joten **Capsule Collider** on poistettava. Sen jälkeen lisätään uusi komponentti valikkokomennolla **Component**

→ **Physics** → **Mesh Collider**. Alustan rajat muutetaan vielä kuperiksi asettamalla **Convex**-valinta päälle.

### 4.1.1 Kamera

Kamerat toimivat pelaajan ”silminä” peliä pelatessa. Kameroita voi olla kentässä useampi, mutta ainakin yksi tarvitaan pelin näyttämiseksi. Useammalla kameralla saadaan aikaiseksi esimerkiksi kahden pelaajan jaettu ruutu ja muita efektejä. (Unity Technologies 2013d.)

Kun uusi kenttä luodaan, siinä on valmiina yksi kamera nimeltä **Main Camera** ja pelin käynnistyessä tämän kameran kuva näkyy. Tason näkymiseksi kameraa siirretään ja pyöritetään, kunnes se osoittaa tasoa. Kameran ollessa valittuna Scene-näkymän vasempaan alanurkaan ilmestyy **Camera Preview** -ikkuna, joka näyttää kameran näkymän.



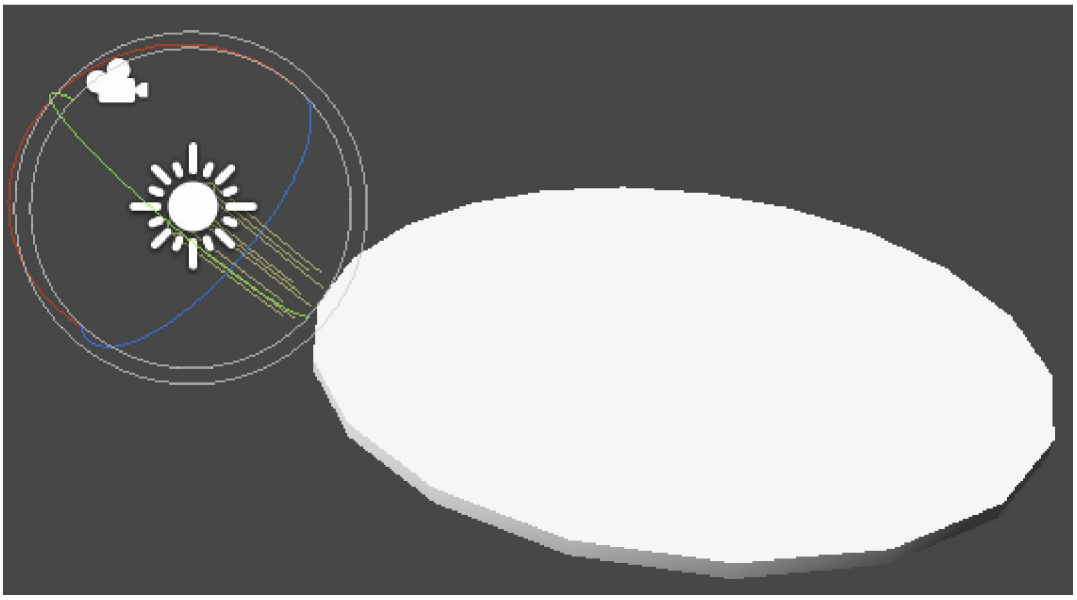

Kuvio 8. Kamera osoittaa alustaan.

Peli käynnistyy **Play**-nappulalla (kuvio 9). Pelissä ei vielä tapahdu mitään, koska toiminnallisuuksia ei ole määritelty peliobjekteille. Pelin ajon voi sammuttaa painamalla **Play**-nappulaa uudestaan.

Kuvio 9. Vasemmalta oikealle: **Play, Pause, Step**.

### 4.1.2 Valaistuksen lisääminen

Kentän esineistä ei voi ilman valoja erottaa kolmiulotteisuutta, joten lisätään suuntavalo valitsemalla **GameObject → Create Other → Directional Light**. Suuntavalo toimii ”aurinkona” kentässä eli se valaisee tasaisesti kaiken (kuvio 10). Suuntavalon sijainnilla ei ole vaikutusta valon voimakkuuteen. Sen voi kuitenkin pyörittää haluamaansa kulmaan.

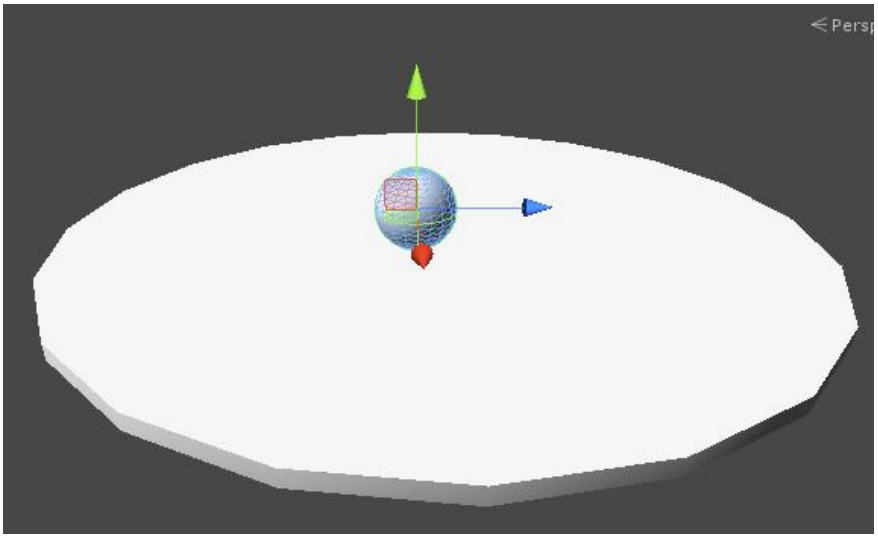


Kuvio 10. Suuntavalo valaisee alustan.

## 4.2 Peliobjektin ohjaus

Ohjattavan objektin luominen vaatii skriptien kirjoittamista. Tässä työssä pelattavaksi hahmoksi valitaan yksinkertainen pallo, jota voi liikutella W-, A-, S- ja D-näppäimillä ja jota kamera seuraa.

Aluksi luodaan pallo komennolla **GameObject → Create Other → Sphere** ja siirretään se pelialustalle. Pallon ei tarvitse olla täysin samalla tasolla alustan kanssa, vaan sen voi jättää vähän koholle kuvion 11 mukaisesti.



Kuvio 11. Pallo alustalla

Pallolle on myös lisättävä Rigidbody, jotta siihen vaikuttaisivat fysiikan lait.

## 4.3 Skriptien luonti

Kentässä on jo tarpeelliset peliobjektit, mutta pelaaja ei voi vaikuttaa niihin. Pelaajan ja peliobjektien välinen vuorovaikutus ja muu logiikka toteutetaan skripteillä. Uusi skripti luodaan valitsemalla **Assets → Create → C# Script**. Skripti ilmestyy Project-näkymään ja sen voi tässä vaiheessa nimetä. Skriptin saa auki kaksoisnapsauttamalla, jolloin MonoDevelop-ohjelma avautuu ja näyttää sen. Skriptiin on kirjoitettu valmiiksi Update- ja Start-metodit sekä tärkeimmät riippuvuudet. *Update*-metodi kutsutaan jokaiselle ruudulle. *Start*-metodi kutsutaan kentän alussa.

### 4.3.1 Kameran suuntaaminen pelihahmoon

Pelaajan kannalta on edullista, että hän näkee pelattavan hahmon koko ajan. Luodaan uusi skripti ja annetaan sille nimeksi **maincamera**. Annetaan skriptille julkinen muuttuja, jonka tyyppi on **Transform**.

```
public class maincamera : MonoBehaviour {
```

```
    public Transform target;
```

**Transform** on jonkin peliobjektin Transform-komponentti. Ennen kuin peli käynnistetään, pitää jokin peliobjekti raahata tämän skriptin ”target”-nimiseen kohtaan. Skripteissä olevia julkisia muuttujia voi muokata Unity-moottorin sisällä.

Seuraavaksi lisätään Update-metodin sisälle **LookAt**-metodi, joka saa skriptin omistajaobjektin kääntymään kohdettaan päin.

```
void Update ()
{
    transform.LookAt(target.position);
}
```

Skripti on nyt valmis ja sen voi rakentaa painamalla F8-näppäintä. Se on enää lisättävä Main Camera -objektin komponentiksi. Kun komponentin **Target**-kohtaan asettaa peliobjektin, kamera seuraa objektia aktiivisesti. Pelin toiminnallisuutta voi testata painamalla Play-nappia. Samaa nappia painamalla peli keskeytyy.

Valmis skripti on esitetty kokonaisuudessaan liitteessä 1.

### 4.3.2 Pelihahmon liikuttaminen

Seuraavaksi luodaan pallolle skripti, jolla palloa voi liikuttaa näppäimistön komennoilla. Luodaan uusi skripti ja annetaan sille nimeksi **player** (Liite 2), sillä tähän skriptiin kirjoitetaan kaikki pelaajan toiminnot.

Koska pelihahmo on pallo, se saadaan liikkeelle lisäämällä pallolle vääntömomenttia tietylle akselille. Näin pallo liikkuu fysiikan lakien mukaisesti, eikä pysty liikkumaan esimerkiksi ollessaan ilmassa tai kun kitkaa on hyvin vähän.

Aluksi skriptiin määritellään julkinen kokonaislukumuuttuja nimeltä **velocity** ja annetaan sille arvoksi **200**. Lisäksi alustetaan yksityinen GameObject-muuttuja, jota tarvitaan pääkameran löytämiseen.

```
public class player : MonoBehaviour {
    public int velocity = 200;

    private GameObject cam;
```

Start-funktiossa asetetaan cam-muuttujalle arvo. Se saadaan etsimällä kentästä Main Camera -niminen peliobjekti.

```
void Start ()
    {
        cam = GameObject.Find("Main Camera");
    }
```

**Update**-metodin jälkeen kirjoitetaan uusi metodi nimeltä **HorizontalMovement**, joka ottaa vastaan nopeuden ja lisää pallolle vääntömomenttia näppäinkomentojen mukaan.

```
void HorizontalMovement(float velocity)
{

    if(Input.GetKey(KeyCode.W))
    {
        rigidbody.AddTorque (transform.right *
        velocity * Time.deltaTime);
    }
    if(Input.GetKey(KeyCode.S))
    {
        rigidbody.AddTorque (transform.right *
        -velocity * Time.deltaTime);
    }
    if(Input.GetKey(KeyCode.A))
    {
        rigidbody.AddTorque (transform.forward *
        velocity * Time.deltaTime);
    }
    if(Input.GetKey(KeyCode.D))
    {
        rigidbody.AddTorque (transform.forward *
        -velocity * Time.deltaTime);
    }
}
```

**RigidBody.AddTorque**-metodi lisää pallon Rigidbody-komponentille vääntöä halutun vektorin suuntaisesti. Metodin parametrin arvo on kerrottu **Time.deltaTime**-arvolla, mikä saa pallon pyörimään aina samaa vauhtia ruudunpäivitysnopeudesta huolimatta. (Unity Technologies 2013c.)

HorizontalMovement-metodia pitää vielä kutsua Update-metodissa, jotta pallo pyörisi aina haluttaessa.

```
void Update ()
{
    HorizontalMovement(velocity);
}
```

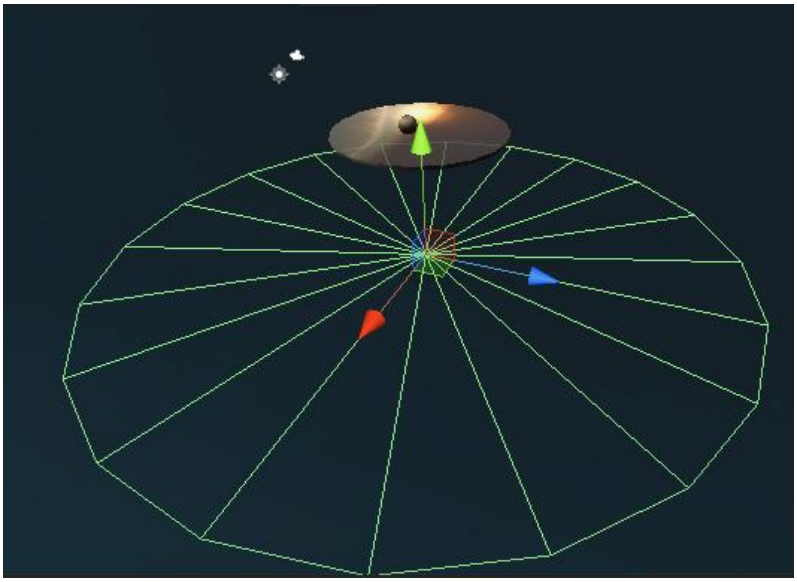
Pelin käynnistäessä pallo reagoi nyt näppäimistön komentoihin ja pyörii suhteessa kameraan. Pallon **Player**-komponentin **Velocity**-arvoa muutettaessa pallon käyttäytyminen muuttuu.

### 4.3.3 Laukaisimet

Peliobjektit voivat toimia näkymättöminä *laukaisimina* (trigger), joita voi käyttää pelimekaniikan apuna. Tässä pelissä laukaisinta tarvitaan esimerkiksi objektin alustalta putoamisen tunnistamiseen.

Laukaisin luodaan tekemällä uusi tyhjä peliobjekti valitsemalla **GameObject → Create Empty** tai painamalla näppäinyhdistelmää **Ctrl+Shift+N**. Annetaan objektille nimeksi **killzone**, sillä kenttä alustetaan peliobjektin osuttua siihen. Killzone-objektille annetaan törmäysrajat valitsemalla **Component → Physics → Mesh Collider**. Rajat voi muuttaa sylinterin muotoisiksi painamalla Inspector-näkymästä **Mesh Collider** -komponentin **Mesh**-kohtaa ja valitsemalla valikosta **Cylinder**. Myös **Is Trigger** -vaihtoehto pitää asettaa päälle, jotta peliobjektit pääsevät törmäysrajoista läpi.

Killzone-objektin kokoa on vielä muutettava ja se on siirrettävä alustan alapuolelle kuvion 12 mukaisesti antamalla sen Position-arvoiksi **0; -8; 0** ja Scale-arvoiksi **40; 0; 40**.



Kuvio 12. Killzone-objekti ja törmäysrajat

Laukaisimelle kirjoitetaan killzone-niminen skripti, joka lisätään laukaisimen komponentteihin. Skripti odottaa peliobjektin osumista törmäysrajoihin. Kun objekti osuu niihin, kenttä alustetaan.

```
public class killzone : MonoBehaviour {

    void OnTriggerEnter()
    {
        Application.LoadLevel(0);
    }
}
```

**OnTriggerEnter** kutsutaan, kun törmäysrajoihin osuu jotain. **Application.LoadLevel** lataa kentän, jonka numero annetaan parametrina. Projektissa on tällä hetkellä vain yksi kenttä, joten sen järjestysnumero on **0**.

### 4.3.4 Metodien kutsuminen muista skripteistä

Skriptien sisällä on mahdollista kutsua metodeita muista skripteistä. Kutsuttavien metodien on oltava julkisia ja niiden sisältävien skriptien on oltava jonkin peliobjektin komponentteja.

```
peliobjekti.GetComponent<skripti> ().Metodi ();
```

### 4.3.5 Skriptin virheiden etsintä

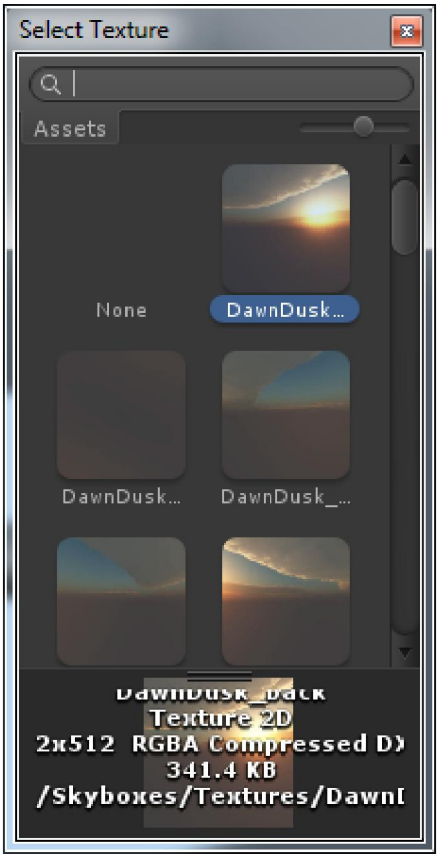
Virheiden etsintä koodista eli *Debuggaus* alustetaan **F5**-näppäimellä MonoDevelop-ohjelmassa, kun Unity ei ole käynnissä. Jos F5-näppäintä painaa Unity-moottorin ollessa käynnissä, tulee virheilmoitus. Unity ja MonoDevelop tukevat *katkaisukohtia (break point)* ja poikkeusten käsittelyä.

## 4.4 Tekstuurit

Tekstuurit ovat osa peliobjektin materiaalia. Muokattavan kentän näkyvillä objekteilla on käytössään perusmateriaali. Objektien tekstuuria voi vaihtaa materiaalin avulla. Uusi materiaali tehdään valitsemalla **Assets → Create → Material**. Materiaali ilmestyy Project-näkymään, jossa sen voi nimetä. Inspector-näkymä näyttää materiaalin ominaisuudet (kuvio 13).


Kuvio 13. Materiaalin ominaisuudet

**Texture**-valinnassa ei tällä hetkellä ole mitään, mutta uuden tekstuurin voi lisätä **Select**-painikeella, joka on tekstuurilaatikon oikeassa reunassa. Ruutuun avautuu valintaikkuna, jossa tekstuuritiedoston voi asettaa materiaalille (kuvio 14).

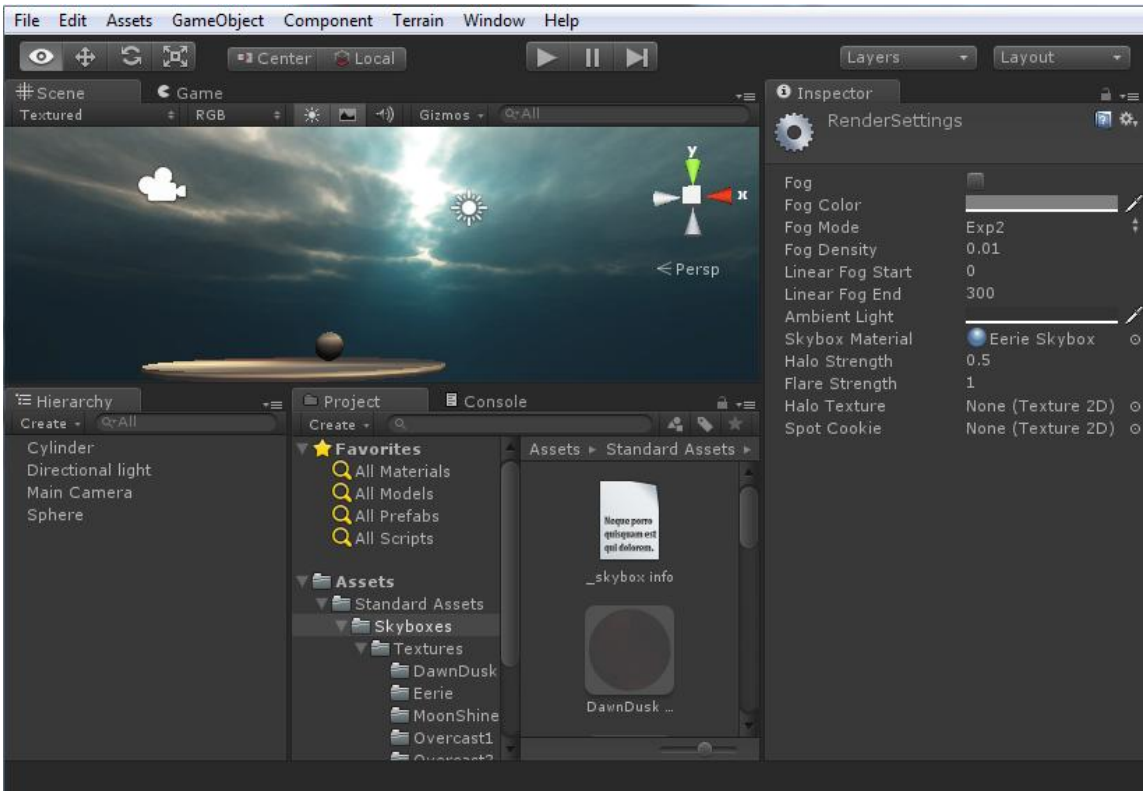



Kuvio 14. Tekstuurin valinta

Kun tekstuuri on valittu, materiaalin voi raahata Project-näkymästä peliobjektin päälle, jolloin peliobjekti vaihtaa väriä.

### 4.5 Skybox-tekniikka

*Skybox* on tekniikka, jolla saadaan lähellä olevat asiat vaikuttamaan äärettömän etäisiltä. Skybox-tekniikkaa käytetään pääasiassa taivaan kuvaamiseen. Unity-moottorissa tekniikka on sisäänrakennettu ja vain tekstuuri pitää lisätä. Tekstuuri lisätään valitsemalla **Edit → Render Settings** ja lisäämällä Skybox-materiaali Inspector-näkymän **Skybox Material** -kohtaan (kuvio 15).



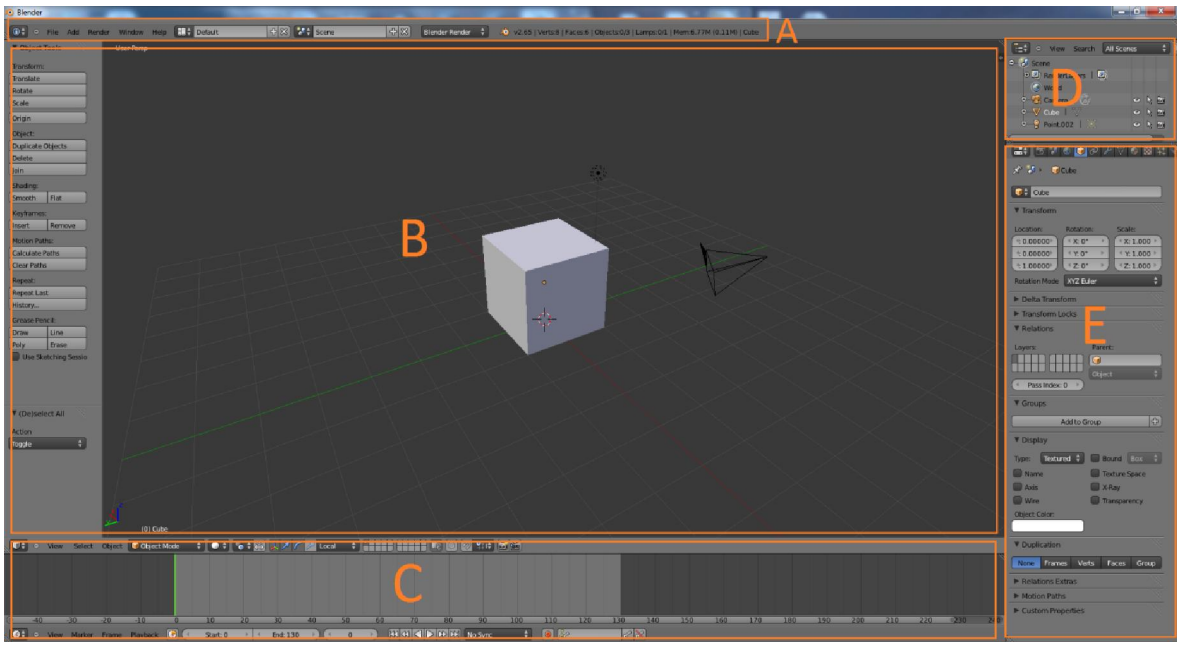

Kuvio 15. Skybox lisättynä kenttään

# 5 BLENDER

Blender on ilmainen avoimen koodin 3D-grafiikan mallinnusohjelma. Ohjelman voi ladata Blender Foundationin sivuilta osoitteesta http://www.blender.org. (Blender Foundation. 2013.)

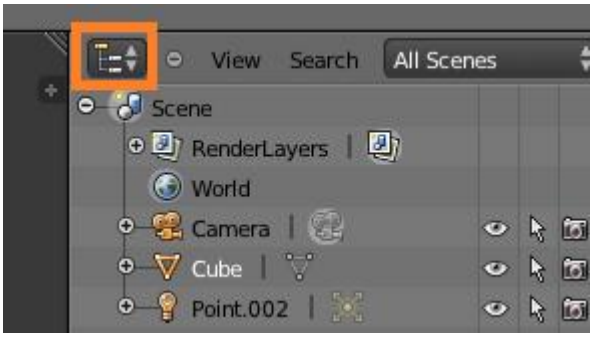
## 5.1 Blender-ohjelman käyttöliittymä

Blender-ohjelman käyttöliittymä koostuu työkalurivistä ja ikkunoista, joiden kokoa ja sisältöä käyttäjä voi muokata vapaasti. Oletuksena ruudulla (kuvio 16) on viisi ikkunaa: **Info-** (kohta A), **3D View**- (kohta B), **Timeline**- (kohta C), **Outliner**- (kohta D) ja **Properties**-ikkunat (kohta E). Nämä ikkunat ovat tärkeimmät 3D-mallinnusta varten.




Kuvio 16. Blender-ohjelman käyttöliittymä oletusasetuksilla

Ikkunoiden kokoa voi kasvattaa tarttumalla reunasta ja raahaamalla. Ikkunoiden sisällön voi valita painamalla niiden vasemmassa ylä- tai alanurkassa olevaa painiketta (kuvio 17).




Kuvio 17. Ikkunan sisällön valitsin

**Info-ikkuna.** Info-ikkunasta hallitaan tiedostoja ja ohjelman asetuksia. Ikkuna muistuttaa valintaikkunaa, joka löytyy useimmista Windows-käyttöjärjestelmän ohjelmista.

**3D-View-ikkuna.** 3D-View on pääikkuna, joka näyttää rakennettavan 3D-mallin. Kuvaa voi liikutella ja pyörittää paremman perspektiivin saamiseksi.

Oletuksena kuvakulman muuttaminen tapahtuu hiiren keskimmäisellä painikkeella. Kuvaa voi pyörittää painamalla hiiren keskimmäistä painiketta ja vetämällä. Kuvaa voi siirtää painamalla yhtäaikaa **Shift**-näppäintä ja hiiren keskimmäistä painiketta. Kuvaa voi loitontaa ja lähentää hiiren rullalla.

Ikkunan kuvakulman saa absoluuttisiin kulmiin painamalla näppäimistön numeronäppäimiä:

- **1** vaihtaa kuvakulman objektin eteen
- **3** vaihtaa kuvakulman objektin oikealle puolelle
- **7** vaihtaa kuvakulman objektin taakse.

Painamalla **Ctrl**-näppäintä yhtä aikaa numeronäppäimen kanssa kuvakulma siirtyy oletusarvon vastakkaiselle puolelle (1 siirtää kuvakulman objektin taakse jne.). Kuvan näkymää voi vaihtaa normaalin ja ortogonaalisen perspektiivin välillä painamalla numeronäppäintä **5**.

**Timeline-ikkuna.** Blender-ohjelmassa on mahdollista tehdä myös animaatioita. Timeline-ikkunaa käytetään animaatoiden luomiseen ja muokkaamiseen.

**Outliner-ikkuna.** Kuten Unity-moottorin Hierarchy-näkymässä, Blender-ohjelmassa tiedostoja käsitellään kenttinä, joissa on objekteja. Outliner-ikkunassa objektit on esitetty hierarkkisessa järjestyksessä.

**Properties-ikkuna.** Properties-ikkuna on verrattavissa Unity-moottorin Inspector-näkymään. Siinä valitun objektin kaikki ominaisuudet näkyvät ja siinä niitä voi muokata.

**3D-kursori.** Blender-ohjelma käyttää apuvälineenä mallinnuksessa tähtäimen näköistä kursoria, joka sijaitsee kolmiulotteisessa avaruudessa. Kursori toimii apuna luodessa uusia objekteja, objektien liikuttamisessa ja perspektiivin kohdistamisessa. Kursorin voi siirtää haluamaansa paikkaan hiiren vasemmalla painikkeella. Kursorin voi siirtää kentän nollapisteeseen painamalla näppäinyhdistelmää **Shift-C**.

## 5.2 Objektin luominen ja muokkaus

Blender-ohjelman käynnistyessä kentän keskellä on oletuksena kuution muotoinen objekti. Objektin voi valita joko hiiren oikealla painikkeella 3D-View-ikkunassa tai hiiren vasemmalla Outliner-ikkunassa, jolloin valitun objektin ääriviivat näkyvät oranssina. 3D-objekteja voi muokata kahdessa eri tilassa: *objektitilassa* (object mode) ja *muokkaustilassa* (edit mode). Objektitilassa objektia voi muokata ja siirrellä, mutta objektin suhteet pysyvät samoina. Muokkaustilassa objektin osia voi valita ja muokata erikseen, muuttaen näin objektin ulkonäköä. (BlenderWiki. 2013.)

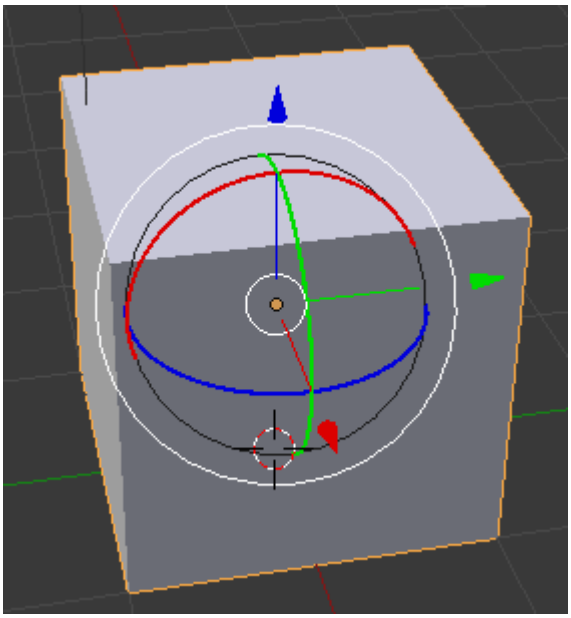
### 5.2.1 Objektitila

Kolmiulotteisia objekteja voi objektitilassa siirrellä, yhdistellä ja muuttaa niiden kokoa. Näiden toimintojen suorittamiseen voi käyttää joko näppäimistöä tai *graafisia manipuloijia* (kuvio 18). Graafisia manipuloijia on kolme: *siirtäjä*

(translate), *kiertäjä* (rotate) ja *skaalaaja* (scale). Manipuloijia voi valita useita kerrallaan pitämällä Shift-näppäintä pohjassa.


Kuvio 18. Graafisen manipuljoijan valitsin.

Objektia muokata tarttumalla hiiren vasemmalla napilla manipuloijan yhdestä akselista kiinni ja raahaamalla objekti toiseen asentoon (kuvio 19).

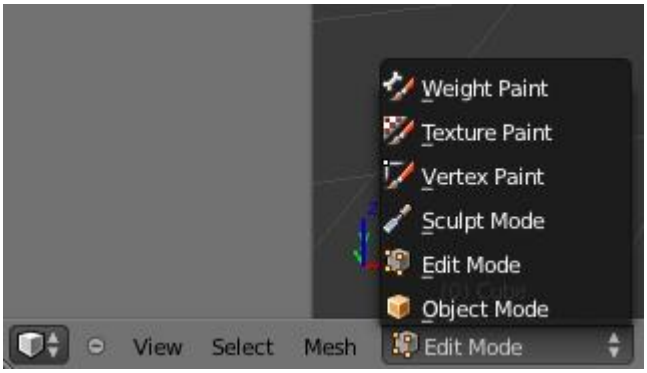


Kuvio 19. Graafiset manipuloijat siirtäjälle ja kiertäjälle.

Objektitilassa manipuloijat vaikuttavat koko objektiin säilyttäen sen suhteet samoina.

### 5.2.2 Muokkaustila

Muokkaustilassa voi valita 3D-objektin osia koko objektin sijaan. Valinnan voi rajata joko *nurkkapisteisiin* (vertex point), *reunoihin* (edge) tai *pintoihin* (face). Muokkaustilan saa päälle joko painamalla sarkainta tai valitsemalla tilalistalta **Edit Mode** (kuvio 20).




Kuvio 20. Muokkaustilan valinta.

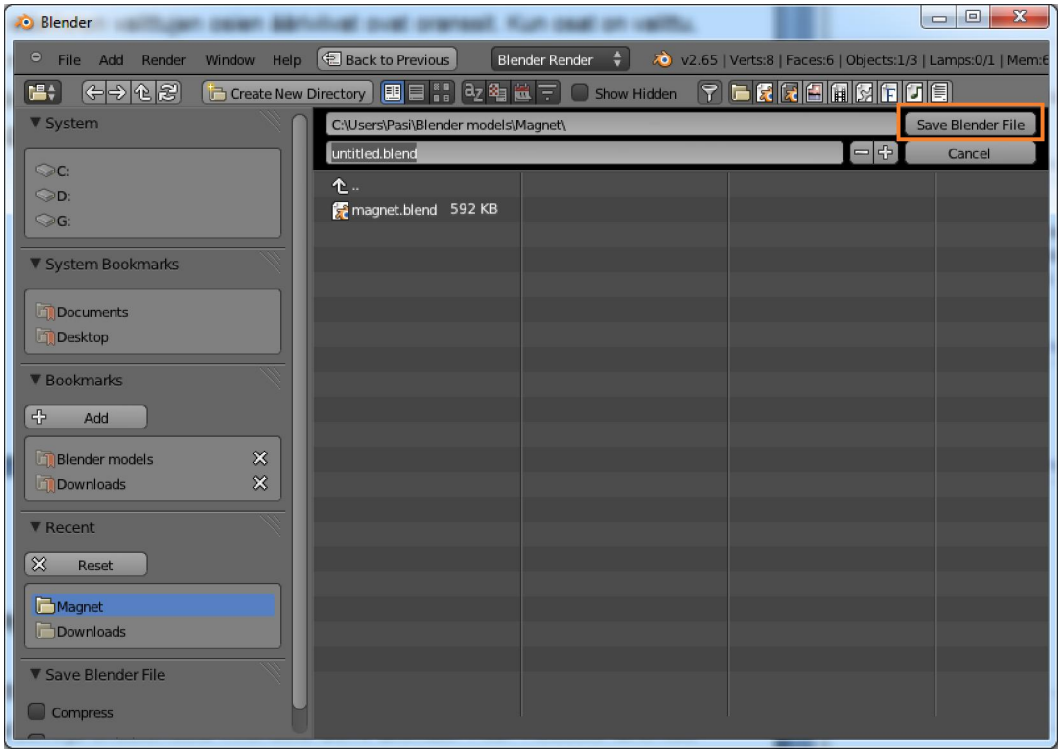
Valintatilan voi vaihtaa napeista valintanapeista (kuvio 21). Valintatiloja voi valita ueita kerrallaan pitämällä Shift-näppäintä pohjassa.

Kuvio 21. Objektin osan valintanapit.

Objektin osia voi valita yksi tai useampi kerrallaan painamalla Shift-näppäimen pohjaan ja painamalla osaa hiiren oikealla napilla. Viimeksi valitun osan ääriviivat ovat valkoiset ja edellisten valittujen osien ääriviivat ovat oranssit. Kun osat on valittu, niitä voi muokata manipuloijien avulla. Tehdyt muutokset voi kumota näppäinyhdistelmällä **Ctrl+Z** tai tehdä uudelleen näppäinyhdistelmällä **Ctrl+Shift+Z**.

### 5.3 3D-objektin tallennus ja siirtäminen Unity-pelimoottoriin

Kun objektin muokkaus on valmis, sen voi tallentaa .blender-tiedostona valitsemalla **Save** ja avautuvasta ikkunasta **Save Blender File** (kuvio 22). Tiedosto tallentuu oletussijaintiin nimellä **untitled.blender**. Sijaintia ja nimeä voi vaihtaa tallennusikkunassa.




Kuvio 22. Tallennusikkuna oletusasetuksilla.

Tiedoston voi tallentaa ylikirjoittamatta vanhaa tiedostoa painamalla **Plus**-nappia nimi-tekstilaatikon oikealla puolella. Tämä kasvattaa tiedoston tunnistenumeroa yhdellä. **Miinus**-nappi vähentää numeroa yhdellä.

Unity osaa useimmiten käsitellä .blender-tiedostoja suoraan. Tiedosto pitää vain tallentaa Unity-projektin **Assets**-kansioon, mistä Unity löytää sen projektin ollessa käynnissä. Sen jälkeen objektin voi siirtää Unity-moottorissa kenttään.

Joissain tapauksissa .blender-tiedoston lukeminen ei välttämättä onnistu. Tällöin mallin voi helpoiten siirtää Unity-kenttään .dae-tiedostomuodossa. Malli tallennetaan .dae-muodossa valitsemalla **File → Export → Collada (default) (.dae)**.

# 6 VIHOLLISTEN LUOMINEN

Videopeleissä on tavallisesti pelaajan lisäksi myös pelihahmoja, joita pelaaja ei ohjaa. Näitä kutsutaan nimellä *NPC* (non-player-character). NPC voi toimia pelaaja kohtaan ystävällisesti, neutraalisti tai vihamielisesti. NPC:n tekeminen poikkeaa päähahmon tekemisestä siten, että sen on toimittava itsenäisesti.

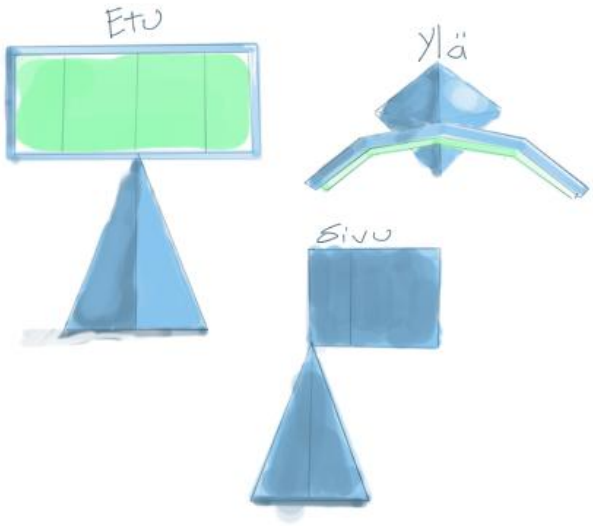
Tässä työssä luodaan vihamielinen NPC, jota pelaajan pitää vältellä.

## 6.1 Vihollisten suunnittelu

Ennen vihollisen luomista sen ulkomuoto ja toiminta on syytä suunnitella. Pelejä ja sen ominaisuuksia suunniteltaessa käytetään usein konseptitaidetta. Konseptitaide on visuaalinen esitys ideoista, ympäristöistä tai hahmoista. Näitä esityksiä käytetään apuna pelin tunnelman ja ilmeen luomisessa. (Roy Nottage. 2009.)

Tämän työn esimerkissä vihollinen on sokea pallo, joka yrittää törmätä pelaajan ohjaamaan palloon. Vihollispallon ”silminä” toimivat *tutkat*, jotka pelaajan pitää kaataa voittaakseen.

**Tutka.** Tutka on paikallaan pysyvä vihollinen, joka ei pysty satuttamaan pelaajaa. Se muodostuu kolmesta osasta: jalasta, lautasesta (lautasantenni) ja valonheittimestä (kuvio 23). Lautanen pyörii jalan päällä ja siinä on kiinni vihreä valonheitin, joka vaihtaa väriä tilanteen mukaan. Jos pelaaja osuu tutkan lautasen näkökenttään, lautanen lukittuu seuraamaan pelaajaa ja ilmoittaa sen viimeisen sijainnin vihollispallolle. Samalla valonheittimen väri vaihtuu punaiseksi. Jos pelaaja liikkuu näköesteen taakse tai tarpeeksi kauas tutkasta, tutka lakkaa seuraamasta pelaajaa ja jatkaa pyörimistä.



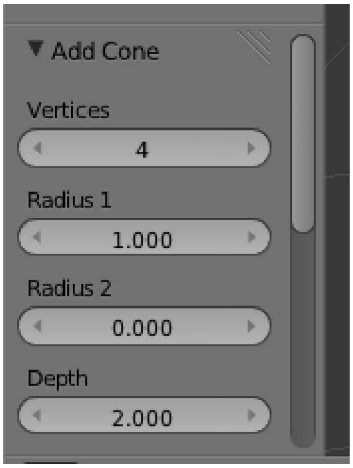

Kuvio 23. Konsepti tutkasta

**Vihollispallo.** Vihollispallo on sokea, eikä liiku pelin alussa. Kun yksi kentän tutkista näkee pelaajan, se ilmoittaa pelaajan sijainnin vihollispallolle. Vihollinen herää ja liikkuu pelaajan viimeiseen ilmoitettuun olinpaikkaan. Jos vihollinen onnistuu koskettamaan pelaajaa, peli on ohi.

## 6.2 Tutkan 3D-malli

Vain tutkan 3D-malli on tehtävä erillisessä ohjelmassa, sillä Unity pystyy luomaan pelkästään primitiivisiä muotoja. Malli rakennetaan Blender-ohjelmassa.

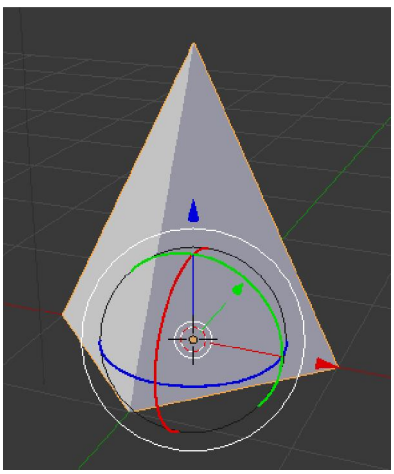
**Tutkan jalka.** Jalkaa varten luodaan kartio painamalla **Shift+A** ja valitsemalla **Mesh → Cone**. Kartion sivujen määrää voi vaihtaa 3D-View-ikkunan vasemmalla puolella olevasta valikosta. Valikko tulee esiin ja menee piiloon näppäimellä **T**. Valikon **Add Cone** -ryhmän **Vertices**-arvoksi asetetaan **4** (kuvio 24).




Kuvio 24. Add Cone -ryhmä

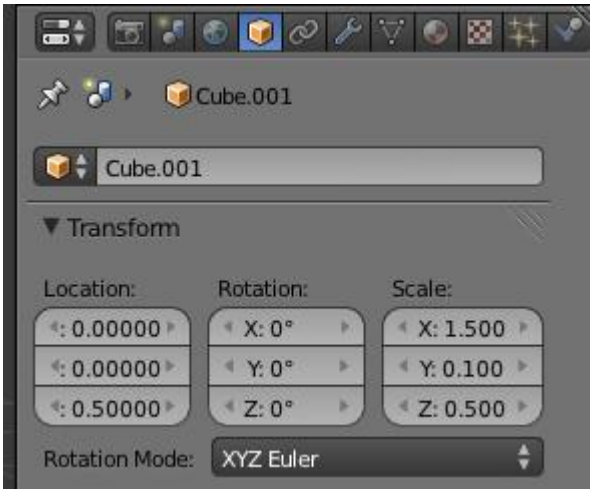
3D-mallin nivelpiste (engl. pivot point) kertoo Unity-pelimoottorille, mikä on mallin nollakohta. Kartion nivelpiste on aluksi sen keskellä, joten se on siirrettävä pohjalle. Kartiota siirretään ensin Z-akselin suuntaisesti ylöspäin joko Properties-ikkunan Object-valikosta tai sinisestä nuolesta vetämällä.

Seuraavaksi painetaan **Ctrl+Shift+Alt+C** ja valitaan valikosta **Origin to 3D Cursor**. Nyt kartion nivelpiste on sen pohjassa, eli Unity-pelimoottorin kenttään lisättäessä se asettuu pintojen päälle oikein (kuvio 25).



Kuvio 25. Kartio ja sen nivelpiste

**Tutkan lautanen ja valonheitin.** Lautasta varten tehdään uusi kuutio-objekti (Cube). Tehty kuutio muokataan laudan muotoiseksi ja siirretään sitä Z-akselin suuntaisesti siten, että se on maan tasalla (kuvio 26).

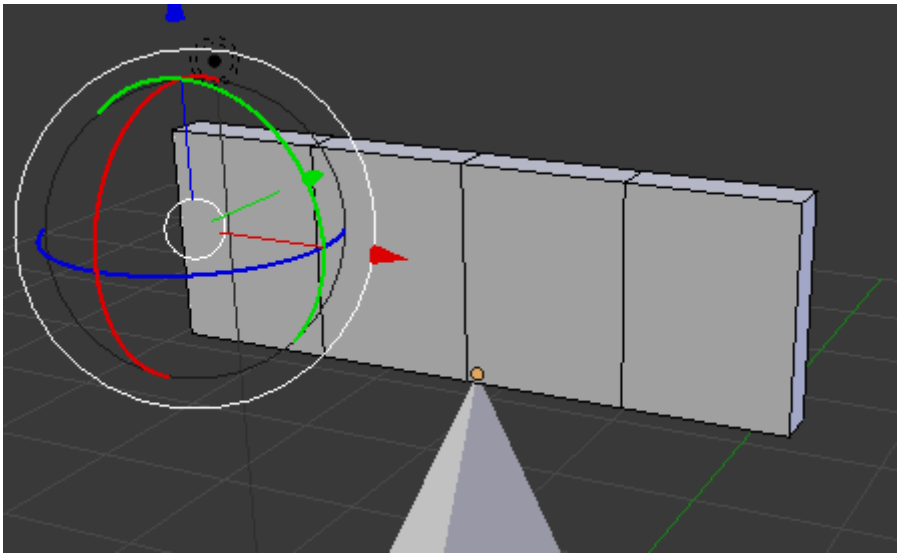



Kuvio 26. Lautasen ulottuvuuksien muutokset

Laudan nivelpiste siirretään pohjalle, kuten kartiolle tehtiin. Lopuksi lauta siirretään kartion päälle.

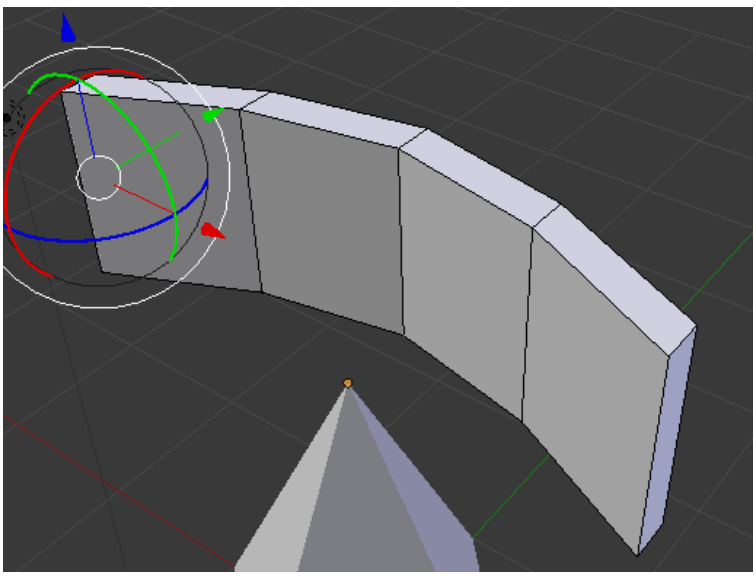
Seuraavaksi tehdään lautasesta kaareva muokkaustilassa. Muokkaustila laitetaan päälle painamalla sarkainta. Lautasta ei voi  vielä kaarevoittaa, koska siinä on liian vähän pintoja. Pinnat voi jakaa osiin lisäämällä niihin *silmukoita* (engl. loop cut). Silmukka lisätään painamalla **Ctrl+**R ja viemällä hiiri lähelle lautasen ylä- tai alareunaa. Silmukan leikkauskohta tulee näkyviin violettina viivana ja leikkaus vahvistetaan hiiren vasemmalla napilla. **Esc**-näppäintä painamalla leikkaus jää objektin keskelle.

Lautanen on vielä leikattava kahdesti siten, että se on jaettu neljään osaan (kuvio 27).



Kuvio 27. Lautasen silmukat

Nyt lautasta on mahdollista muokata vapaammin haluttuun muotoon. Lautasen keskimmäisen silmukan voi valita hiiren vasemmalla, kun **Alt**-näppäin on pohjassa ja valinta on rajattu reunoihin. Seuraavaksi siirretään silmukkaa Y-akselin suunnassa. Sitten valitaan ulompireunaiset silmukat pitämällä **Alt**- ja **Shift**-näppäimiä pohjassa ja valitsemalla molemmat silmukat hiiren oikealla napilla. Silmukoita siirrellään, kunnes lautasen muoto on tarpeeksi kaareva (kuvio 28).

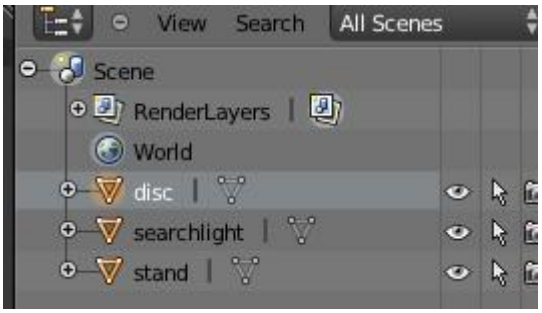


Kuvio 28. Kaarevaksi muokattu lautanen

Lautanen on siirrettävä Y-akselin suuntaisesti, kunnes se on kartion päällä. Tämä onnistuu helpoiten objektitilassa.

Tutkan valonheitin on käytännössä pienempi versio lautasesta. Lautasesta tehdään kopio valitsemalla se ja painamalla **Shift+D**. Kopiota siirretään hieman pois päin lautasesta Y-akselilla **Y**-näppäimellä. Kopion paikka varmistetaan hiiren vasemmalla napilla ja sitä kutistetaan **S**-näppäimellä.

Lopuksi osat valitaan hiiren oikealla napilla Outliner-ikkunassa ja nimetään uudelleen pikavalikon **Rename**-komennolla (kuvio 29).

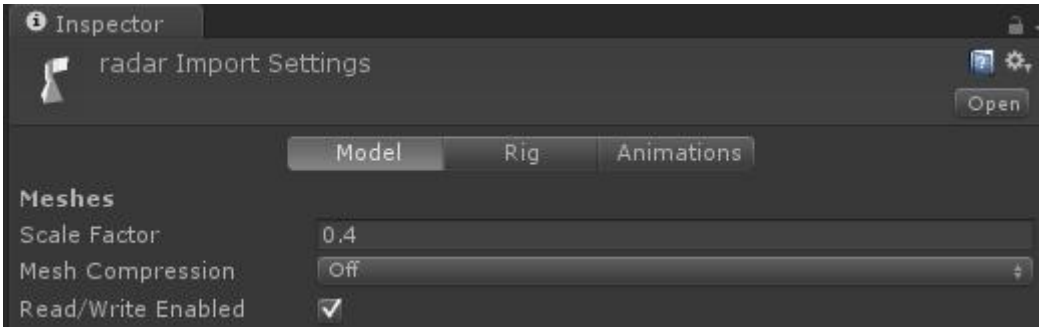



Kuvio 29. Osien uudet nimet

### 6.2.1 Tutkan tuominen Unity-ympäristöön

Ennen objektien tuomista on ylimääräiset objektit, kuten kamerat ja valot, poistettava. Objektit voi poistaa Outliner-ikkunassa pikavalikon **Delete**-komennolla.

Varmin keino tuoda tutkan malli Unity-ympäristöön on tallentaa se .dae-tiedostona. Tallennus tapahtuu valikkokomennolla **File → Export → Collada (default) (.dae)** ja tiedoston nimeksi annetaan **radar.dae**. Sitten avataan Unity ja valitaan hiiren vasemmalla Project-näkymästä tallennettu malli. Inspector-näkymä näyttää mallin tiedot. Tällä hetkellä malli on liian suuri, joten pienennetään sen kokoa kirjoittamalla **Scale Factor** -kentän arvoksi **0.4** (kuvio 30) ja sen jälkeen valitaan **Apply**.




Kuvio 30. Tutkan tuontiasetukset

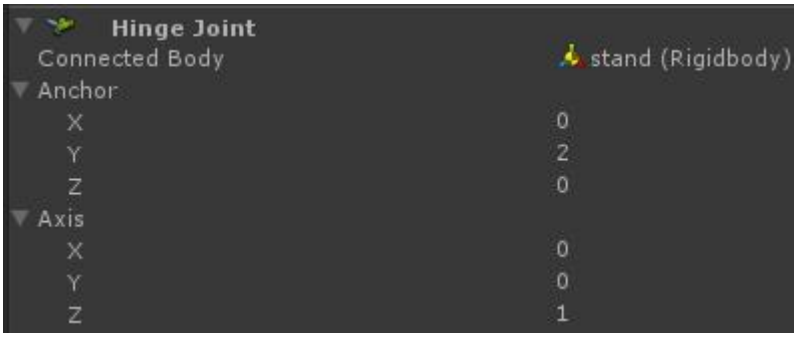
Nyt tutkan voi raahata Project-näkymästä kentän alustalle. Hierarchy-näkymään avautuu uusi objekti nimeltä **radar** ja sillä on kaksi lasta. Jos mallia muuttaa Blender-ohjelmassa ja ylikirjoittaa vanhan mallin uudella, Unity päivittää kaikki kentän sisällä olevat mallit.

**Tutkan värit.** Tutkan jalalle ja lautaselle voi määrittää uuden värin tekemällä tutkalle uuden materiaalin. Materiaalin väriksi asetetaan sininen. Valonheittimelle on tehtävä uusi itsevalaiseva materiaali. Tämä onnistuu vaihtamalla uuden materiaalin **Shader**-ominaisuuden arvoksi **Self-Illumin/Bumped Diffuse**.

### 6.2.2 Fysiikkamallinnuksen lisääminen tutkaan

Tutkasta on tehtävä vuorovaikutteinen objekti, joka tottelee painovoimaa. Sille pitää siis lisätä Rigidbody- ja Mesh Collider -komponentit.

Lautasen on pystyttävä liikkumaan vain Y-akselilla tutkan jalan päällä, joten sille on annettava nivel. Valitaan lautanen ja valikosta **Component → Physics → Hinge Joint**. Tutkan jalka vedetään Hierarchy-näkymästä Hinge Joint -komponentin **Connected Body** -kohtaan. Nivelen suunta ja paikka on vielä muutettava vaihtamalla **Anchor**-kohdan **Y**-arvo numeroksi **2** ja **Axis**-kohdan **Z**-arvo numeroksi **1** (kuvio 31). Nivel näkyy pienenä oranssina nuolena Scene-näkymässä.




Kuvio 31. Nivelen asetukset

## 6.3 Vihollisten tekoäly

Tutka ja vihollispallo toimivat yhdessä siten, että tutka kertoo vihollispallolle pelaajaan sijainnin ja vihollispallo hyökkää sinne.

### 6.3.1 Tutka ja pelaajan etsiminen

Aluksi on saatava lautanen pyörimään ja etsimään pelaajaa. Jotta valonheitin pyörisi samassa tahdissa lautasen kanssa, se on asetettava lautasen lapseksi tarttumalla ja pudottamalla Hierarchy-näkymään.

Uusi skripti **radarAI** pyörittää tutkan lautasta.

```
public GameObject target;
Transform disc;
Transform searchlight;

// Use this for initialization
void Start () {
    disc = transform.FindChild("disc");
    searchlight = transform.FindChild("disc/searchlight");
}

// Update is called once per frame
void Update () {
    LookForTarget();
}

void LookForTarget ()
{
    disc.rigidbody.angularDrag = 1;
    disc.rigidbody.AddTorque (0, 10 * Time.deltaTime, 0);
}
```

Skripti alustaa muuttujat **disc** ja **searchlight**, jotka pelin alussa asetetaan tutkan lautaseksi ja valonheittimeksi. AddTorque-metodi lisää tasaisesti lautasen vääntömomenttia, jolloin sen pyörimiseen voi vaikuttaa esimerkiksi pelaajan toiminnalla. **Target**-muuttuja on peliobjekti, jota tutka etsii.

Seuraavaksi määritellään, milloin tutka näkee pelaajan. Määrittely tehdään uudella metodilla **InsideFOV**, joka laskee kahden vektorin välisen kulman. Ensimmäinen vektori lähtee valonheittimestä eteenpäin ja toinen kulkee valonheittimestä kohteeseen. Jos vektorien kulma on suurempi kuin puolet tutkan näkökentästä (engl. field-of-vision), kohde on tutkan näkökentän sisällä.

```
public float fov = 90.0f;
bool InsideFOV ()
{
    Vector3 vec1 = -searchlight.transform.up;
    Vector3 vec2 = target.transform.position - sear-
chlight.transform.position;
    float angle = Vector3.Angle (vec1, vec2);

    if (angle <= fov / 2)
        return true;
    return false;
}
```

Tutkan valonheitin vaihtaa väriä punaiseksi kohteen ollessa näkyvissä.

```
void Update () {
    LookForTarget();
    if (InsideFOV())
        searchlight.renderer.material.color = Color.red;
    else
        searchlight.renderer.material.color = Color.green;
}
```

Nyt tutka näkee pelaajan esteiden läpi, mutta esteet tutkan ja pelaajan välissä voi tarkistaa **SeesTarget**-metodilla.

```
bool SeesTarget ()
{
    RaycastHit hit;
    if (Physics.Raycast (searchlight.transform.position,
        target.transform.position
        - searchlight.transform.position,
```

```
        out hit, 10)) {
        if (hit.collider.gameObject.name == target.name)
            return true;
        return false;
    }
    return false;
}
```

Valonheittimestä lähetetään säde, joka palauttaa osuessaan törmäysobjektin nimen. Jos nimi on sama kuin kohteella, säde pääsee kohteen luo suoraan. Kun tämä tarkistus lisätään Update-metodiin, tutka näkee objektin vain kun se on kääntynyt kohdetta päin ja esteitä ei ole tiellä.

Tutkan skriptin lopullinen muoto on esitetty liitteessä 3.

### 6.3.2 Vihollispallo

Vihollispallo on massaltaan ja kooltaan identtinen pelaajan pallon kanssa. Se odottaa pelin alussa tutkalta koordinaatteja. Koordinaatit saatuaan pallo liikkuu niiden osoittamaan paikkaan ja odottaa uusia koordinaatteja.

Tarvitaan uusi skripti **enemyball**. Se alustaa uuden peliobjektin **targeting**, joka auttaa palloa liikkumaan oikeaan suuntaan. Tämä on tarpeellinen siksi, että pallon koordinaatit pyörivät pallon mukana.

```
GameObject targeting;
void Start()
{
    targeting = new GameObject("targeting");
}
```

Pallon ei tarvitse osata muuta kuin liikkua eteenpäin, koska **LookAt**-metodilla se osoittaa aina oikeaan suuntaan kun koordinaatit on annettu.

```
float velocity = 200.0f;
void MoveForward()
{
    rigidbody.AddTorque (targeting.transform.right * velocity *
Time.deltaTime);
}
```

Pallo tarvitsee vielä julkisen metodin, jota tutka kutsuu. Tällä metodilla pallolle annetaan koordinaatit, johon pallo lähtee liikkumaan.

```
Vector3 lastPosition;
bool active;
public void SetLastPosition(Vector3 lastpos)
{
    lastPosition = lastpos;
    active = true;
}
```

Lopuksi päivitetään Update-metodi.

```
void Update () {
    targeting.transform.position = transform.position;
    if (active)
    {
        targeting.transform.LookAt(lastPosition);
        MoveForward();
    }
}
```

Nyt tutkan on kutsuttava SetLastPosition-metodia aina pelaajan nähdessään.

```
public GameObject enemyball;

void Update () {
    LookForTarget();
    if (InsideFOV() && SeesTarget())
    {
        enemyball.GetComponent<enemyball>().SetLastPosition(
        target.transform.position);
        searchlight.renderer.material.color = Color.red;
    }
    else
        searchlight.renderer.material.color = Color.green;
}
```

Valmiit skriptit voidaan siirtää peliobjekteille. Ensin tehdään uusi pallo, jolla on RigidBody-komponentti. Pallolle annetaan enemyball-skripti ja tutkalle radarAI-skripti. Raahataan tutkan skriptin **Target**-kohtaan pelaajan pallo ja **Enemyball**-kohtaan vihollispallo.

### 6.4 Pelaajan ja vihollisen törmäys

Pelin tarkoituksena on vältellä vihollispalloa, joten siihen törmäämisestä on rankaistava pelaajaa. Törmäykset käsitellään skripteissä **OnCollisionEnter**-metodilla. Lisätään metodi player-skriptiin ja alustetaan kenttä, jos törmääjä on vihollispallo.

```
public void OnCollisionEnter(Collision collision)
{
    if (collision.gameObject.name == "enemyball")
        Application.LoadLevel(0);
}
```

Vihollispallon skriptin lopullinen muoto on esitetty liitteessä 4.

# 7 KÄYTTÖLIITTYMÄN LUONTI

Graafinen käyttöliittymä antaa pelaajan hallita pelin toimintoja kuvien avulla ilman tekstin kirjoittamista. Sen avulla voi esimerkiksi ladata kentän uudelleen tai säätää äänenvoimakkuutta. Unity-pelimoottorissa käyttöliittymät piirretään skriptien avulla. Unity-moottorissa on valmiiksi tyylit napeille ja muille käyttöliittymän ohjaimille, mutta omiakin tyylejä voi luoda.

## 7.1 Taukovalikko

Lähes kaikki pelit voidaan keskeyttää ja sen ominaisuuksia säätää kesken pelin. Tätä varten ruutuun ilmestyy *taukovalikko* (engl. Pause Menu) ja peli pysähtyy.

Pelin aikaa hallitaan **Time.timeScale**-muuttujalla. Kaksiulotteisten graafisten elementtien lisääminen kameran kuvaan tapahtuu skriptin **OnGui**-metodilla.

### 7.1.1 Pelin pysäyttäminen

Pelin pysähtymistä varten **camera.cs**-skriptiin kirjoitetaan metodit **Pause** ja **UnPause** sekä pelin tilaa tarkasteleva julkinen **bool**-muuttuja **paused**. Pause-metodi pysäyttää pelin ajan ja näyttää kursorin. UnPause-metodi asettaa ajan kulun normaaliksi ja piilottaa kursorin.

```
bool paused;

public void Pause()
{
    Time.timeScale = 0; paused = true;
    Screen.showCursor = true;
    Screen.lockCursor = false;
}
public void UnPause()
{
    Time.timeScale = 1; paused = false;
    Screen.showCursor = false;
    Screen.lockCursor = true;
}
```

**Time.timeScale**-muuttuja kertoo pelin nopeuden. Peli kulkee normaalilla vauhdilla, kun muuttujan arvo on **1**. Peliä voi nopeuttaa ja hidastaa muuttamalla tätä arvoa.

**Update**-metodiin on vielä lisättävä tarkistus, joka odottaa Esc-näppäimen painallusta.

```
void Update () {
        transform.LookAt(target.position);
        if(Input.GetKeyDown(KeyCode.Escape)) {
            if(!paused) {
                Pause ();
            }
            else {
                UnPause();
            }
        }
    }
```

Peliä ajettaessa Esc-näppäin pysäyttää pelin ja näyttää kursorin, jos se on piilotettu. Kun näppäintä painaa uudestaan, pelin käynnistyy ja kursori piiloutuu.

### 7.1.2 Valikon piirtäminen

Valikon näyttämiseksi **camera.cs**-skriptiin tehdään **OnGui**-metodi. Tämä metodi kutsutaan jokaiselle kuvalle (frame) samaan tapaan kuin **Update**-metodia. OnGui-metodin sisällä kutsutaan metodia **PauseMenu**, kun **paused**-ehto täyttyy.

```
void OnGUI()
{
    if (paused) {
        PauseMenu();
    }
}
```

PauseMenu-metodissa piirretään laatikko ruudun keskelle ja laatikon sisälle laitetaan nappulat **PauseMenuButtons**-metodilla. Kun **GUI.Button**-metodin piirtämää nappia painetaan, metodi palauttaa arvon **true**. Metodille annetaan

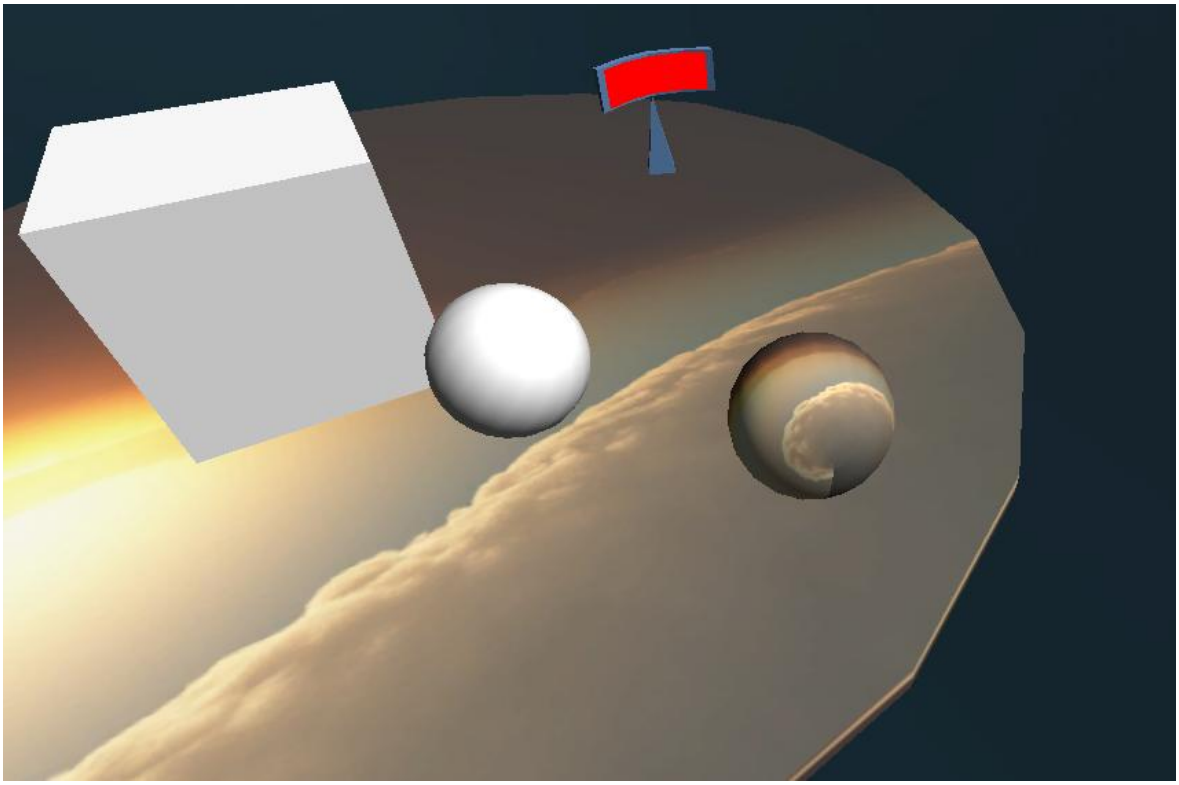
arvoiksi **Rect**-muuttuja, joka kertoo napin sijainnin ja koon, sekä napin nimeksi valittu merkkijono.

```
void PauseMenu ()
{
    Rect pauseBox = new Rect (Screen.width / 2 - 100,
        Screen.height / 2 - 120,
        200, 240);
    GUI.Box (pauseBox, "Paused");

    PauseMenuButtons (pauseBox);
}
```

```
void PauseMenuButtons (Rect pauseBox)
{
    Vector2 button =
    new Vector2 (pauseBox.width / 2,
        pauseBox.height / 4);

    if (GUI.Button (new Rect (pauseBox.xMin
        + (pauseBox.width - button.x) / 2,
        pauseBox.yMin + button.y
        * 1, button.x, button.y), "Return")) {
            UnPause ();
        }
    if (GUI.Button (new Rect (pauseBox.xMin
        + (pauseBox.width - button.x) / 2,
        pauseBox.yMin + button.y
        * 2 + 5, button.x, button.y), "Restart")) {
           UnPause ();
            Application.LoadLevel (0);
        }
}
```

Ensimmäinen nappi on nimeltään ”Return” ja se jatkaa peliä. Toinen nappi ”Restart” alustaa kentän.

Peli on nyt valmis pelattavaksi.



Kuvio 32. Peli vauhdissa

# 8 YHTEENVETO

Tässä työssä pohdittiin videopelin kehittämisen vaiheita ja esiteltiin kolme pelimoottoria. Kehitysympäristöksi valitulla Unity3D-pelimoottorilla tehtiin yksinkertainen peli, jonka tekovaiheet dokumentoitiin. Pelimoottorin lisäksi pelin tekemisessä käytettiin Blender-mallinnusohjelmaa. Ohjelman avulla luotiin pelihahmo peliprojektiin ja luomisen vaiheet käytiin läpi.

Työn tavoitteena oli ohjeistaa pelinkehityksen ensikertalaista rakentamaan videopeli Unity3D-pelimoottorin avulla.

# LÄHTEET


Blender Foundation. 2013. [www-dokumentti]. Blender.org. [Viitattu 13.3.2013]. Saatavissa: http://www.blender.org/

BlenderWiki. 2013. [www-dokumentti]. Wiki.Blender.org. [Viitattu 24.3.2013]. Saatavissa: http://wiki.blender.org/index.php/Doc:2.6/Manual/Modeling/Meshes

Crosby, T. 2011. How Making a Video Game Works?. [www-dokumentti]. HowStuffWorks.com. [Viitattu 11.4.2013]. Saatavissa: http://www.howstuffworks.com/making-a-video-game.htm

CryTek. 2013. Licensing. [www-dokumentti]. MyCryEngine.com. [Viitattu 11.4.2013]. Saatavissa: http://mycryengine.com/?conid=3

Epic Games. 2013. Licensing. [www-dokumentti]. UnrealEngine.com. [Viitattu 11.4.2013]. Saatavissa: http://www.unrealengine.com/en/licensing/

McKleinfeld, D. 2012. Indie game developers rally behind cheap-to-use Unity Engine at Unite 2012. [www-dokumentti]. DigitalTrends.com. [Viitattu 12.4.2013]. Saatavissa: http://www.digitaltrends.com/computing/is-the-unity-engine-ready-for-the-speedway/

Roy Nottage. 2009. What is a Concept Artist? [www-dokumentti]. Royzo.co.uk. [Viitattu 26.3.2013]. Saatavissa: http://www.royzy.co.uk/2009/03/what-is-a-concept-artist.html

Unity Technologies. 2013a. Fast Facts. [www-dokumentti]. Unity3D.com. [Viitattu 23.1.2013]. Saatavissa: http://unity3d.com/company/public-relations/

Unity Technologies. 2013b. Products. [www-dokumentti]. Unity3D.com. [Viitattu 11.4.2013]. Saatavissa: https://store.unity3d.com/

Unity Technologies. 2013c. Script Reference. [www-dokumentti]. Unity3D.com. [Viitattu 13.3.2013]. Saatavissa: http://docs.unity3d.com/Documentation/ScriptReference/

Unity Technologies. 2013d. Cameras. [www-dokumentti]. Unity3D.com. [Viitattu 11.3.2013]. Saatavissa: http://docs.unity3d.com/Documentation/Manual/Cameras.html

Unity Technologies. 2013e. Materials. [www-dokumentti]. Unity3D.com. [Viitattu 11.3.2013]. Saatavissa: http://docs.unity3d.com/Documentation/Manual/Materials.html

Unity Technologies. 2013f. Light. [www-dokumentti]. Unity3D.com. [Viitattu 10.3.2013]. Saatavissa: http://docs.unity3d.com/Documentation/Components/class-Light.html

Ward, J. 2008. What is a Game Engine? [www-dokumentti]. GameCareerGuide.com. [Viitattu 22.1.2013]. Saatavissa: http://www.gamecareerguide.com/features/529/what_is_a_game_.php

# LIITTEET

**Liite 1: camera.cs**

**Liite 2: player.cs**

**Liite 3: radarAI.cs**

**Liite 4: enemyball.cs**

## Liite 1: camera.cs

```
using UnityEngine;
using System.Collections;

public class camera : MonoBehaviour {
    public Transform target;
    bool paused;
    // Use this for initialization
    void Start () {
        Screen.showCursor = false;
        Screen.lockCursor = true;
    }

    // Update is called once per frame
    void Update () {
        transform.LookAt(target.position);
        if(Input.GetKeyDown(KeyCode.Escape)) {
            if(!paused) {
                Pause ();
            }
            else {
                UnPause();
            }
        }
    }

    void OnGUI()
    {
        if (paused) {
            PauseMenu();
        }
    }

    void PauseMenu ()
    {
        Rect pauseBox = new Rect (Screen.width / 2 - 100,
            Screen.height / 2 - 120,
            200, 120);

        GUI.Box (pauseBox, "Paused");

        PauseMenuButtons (pauseBox);
    }

        void PauseMenuButtons (Rect pauseBox)
    {
           Vector2 button = new Vector2 (pauseBox.width / 2, pause-
           Box.height / 4);

           if (GUI.Button (new Rect (pauseBox.xMin + (pauseBox.width -
           button.x) / 2,
            pauseBox.yMin + button.y * 1, button.x, button.y), "Return"))
           {
            UnPause ();
           }
        if (GUI.Button (new Rect (pauseBox.xMin + (pauseBox.width - but-
        ton.x) / 2,
```

```
            pauseBox.yMin + button.y * 2 + 5, button.x, button.y), "Res-
            tart")) {
                UnPause();
                Application.LoadLevel (0);
            }
        }

        public void Pause()
        {
            Time.timeScale = 0; paused = true;
            Screen.showCursor = true;
            Screen.lockCursor = false;
        }
        public void UnPause()
        {
            Time.timeScale = 1; paused = false;
            Screen.showCursor = false;
            Screen.lockCursor = true;
        }
}
```

## Liite 2: player.cs

```
using UnityEngine;
using System.Collections;

public class player : MonoBehaviour {
    public int velocity = 200;

    private GameObject cam;

    // Use this for initialization
    void Start () {
        cam = GameObject.Find("Main Camera");
    }

    // Update is called once per frame
    void Update () {
        HorizontalMovement(velocity);
    }

    void HorizontalMovement(float velocity)
    {
        if(Input.GetKey(KeyCode.W))
        {
            rigidbody.AddTorque (cam.transform.right * velocity *
Time.deltaTime);
        }
        if(Input.GetKey(KeyCode.S))
        {
            rigidbody.AddTorque (cam.transform.right * -velocity *
Time.deltaTime);
        }
        if(Input.GetKey(KeyCode.A))
        {
            rigidbody.AddTorque (cam.transform.forward * velocity *
Time.deltaTime);
        }
        if(Input.GetKey(KeyCode.D))
        {
            rigidbody.AddTorque (cam.transform.forward * -velocity *
Time.deltaTime);
        }
    }

    public void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.name == "enemyball")
            Application.LoadLevel(0);
    }
}
```

## Liite 3: radarAI.cs

```
using UnityEngine;
using System.Collections;

public class radarAI : MonoBehaviour {

    public float fov = 90.0f;
    public GameObject target;
    public GameObject enemyball;

    Transform disc;
    Transform searchlight;

    // Use this for initialization
    void Start () {
        disc = transform.FindChild("disc");
        searchlight = transform.FindChild("disc/searchlight");
    }

    // Update is called once per frame
    void Update () {
        LookForTarget();
        if (InsideFOV() && SeesTarget())
        {
            enemy-
ball.GetComponent<enemyball>().SetLastPosition(target.transform.position)
;
            searchlight.renderer.material.color = Color.red;
        }
        else
            searchlight.renderer.material.color = Color.green;
    }

    void LookForTarget ()
    {
        disc.rigidbody.angularDrag = 1;
        disc.rigidbody.AddTorque (0, 10 * Time.deltaTime, 0);
    }

    bool InsideFOV ()
    {
        Vector3 vec1 = -searchlight.transform.up;
        Vector3 vec2 = target.transform.position - sear-
chlight.transform.position;
        float angle = Vector3.Angle (vec1, vec2);

        if (angle <= fov / 2)
            return true;
        return false;
    }

    bool SeesTarget ()
    {
        RaycastHit hit;
```

```
        if (Physics.Raycast (searchlight.transform.position, tar-
get.transform.position - searchlight.transform.position, out hit, 10)) {
            if (hit.collider.gameObject.name == target.name)
                return true;
            return false;
        }
        return false;
    }
}
```

## Liite 4: enemyball.cs

```
using UnityEngine;
using System.Collections;

public class enemyball : MonoBehaviour {

    Vector3 lastPosition;
    bool active;
    float velocity = 200.0f;
    GameObject targeting;

    void Start()
    {
        targeting = new GameObject("targeting");
    }

    // Update is called once per frame
    void Update () {
        targeting.transform.position = transform.position;
        if (active)
        {
            targeting.transform.LookAt(lastPosition);
            MoveForward();
        }
    }

    public void SetLastPosition(Vector3 lastpos)
    {
        lastPosition = lastpos;
        active = true;
    }

    void MoveForward()
    {
        rigidbody.AddTorque (targeting.transform.right * velocity *
Time.deltaTime);
    }
}
```